Digital Design & Computer Architecture
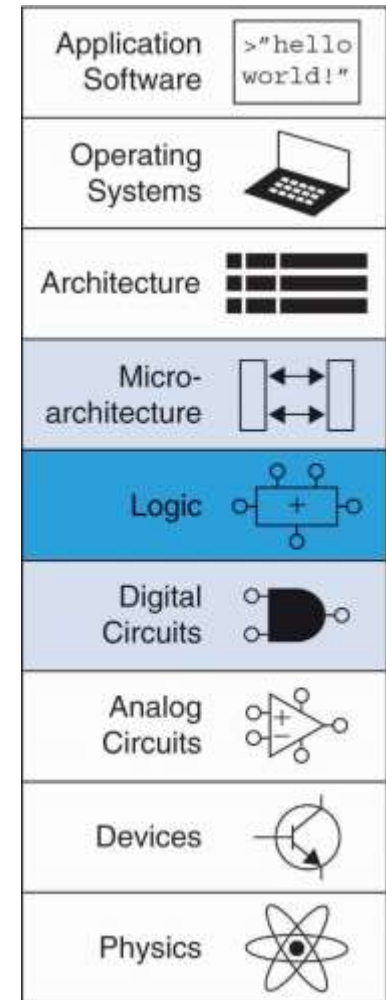
Sarah Harris & David Harris

Chapter 5:
Digital Building Blocks

Chapter 5 :: Topics

- **Introduction**
- **Arithmetic Circuits**
- **Number Systems**
- **Sequential Building Blocks**
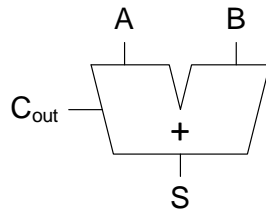- **Memory Arrays**
- **Logic Arrays**

Introduction

- **Digital building blocks:**
  - Gates, multiplexers, decoders, registers, arithmetic circuits, counters, memory arrays, logic arrays

- **Building blocks demonstrate hierarchy, modularity, and regularity:**
  - Hierarchy of simpler components
  - Well-defined interfaces and functions
  - Regular structure easily extends to different sizes

- **We'll use these building blocks in Chapter 7 to build a microprocessor**

Adders

1-Bit Adders

**Half
Adder**

**Full
Adder**

A   B
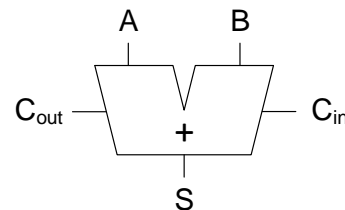
C_{out} —   + —

S

A   B

C_{out} —   + — C_{in}

S

A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

S   =
C_{out} =

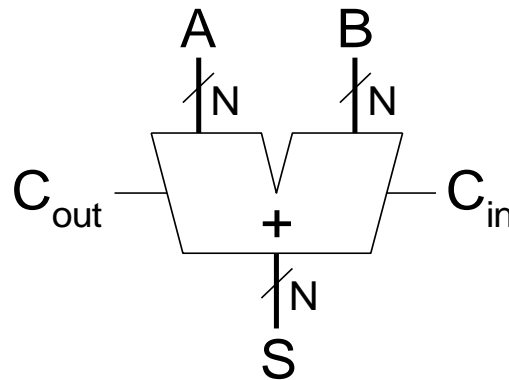| C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

S   =
C_{out} =

Multibit Adders: CPAs

- Types of carry propagate adders (CPAs):
  - **Ripple-carry**   (slow)
  - **Carry-lookahead**   (fast)
  - **Prefix**   (faster)
- Carry-lookahead and prefix adders faster for large adders but require more hardware
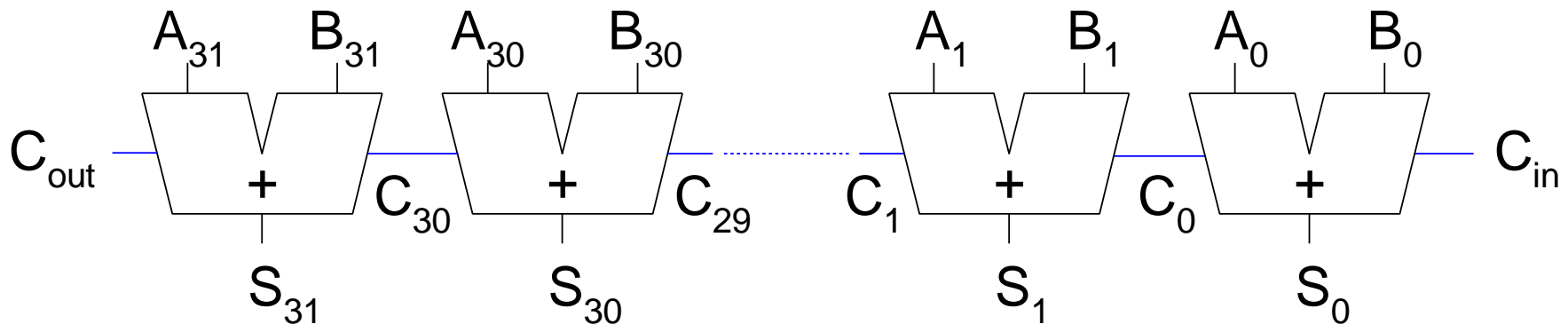
Symbol

Ripple Carry Addition

Ripple-Carry Adder
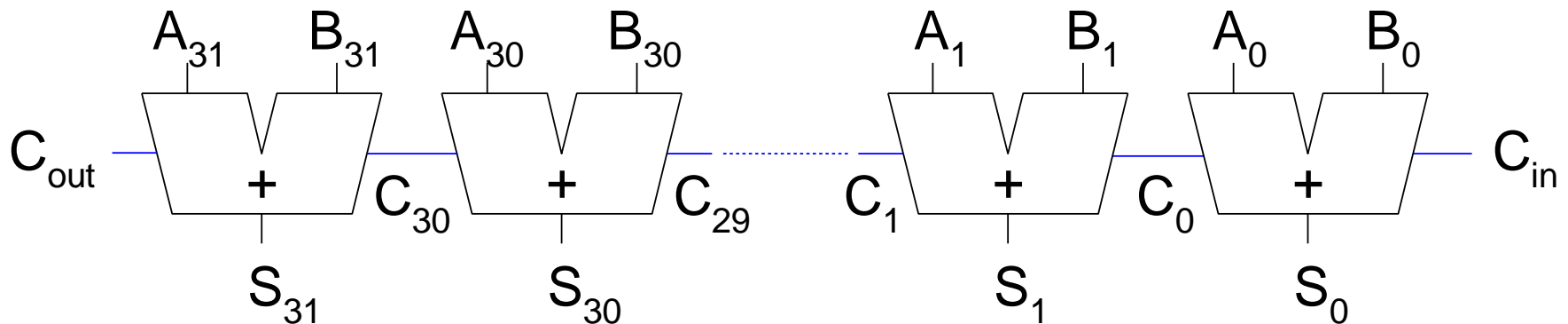
- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**

$$t_{\mathrm{ripple}} =$$

where t_{FA} is the delay of a 1-bit full adder

Carry Lookahead Addition

Carry-Lookahead Adder

Compute C_{out} for k-bit blocks using *generate* and *propagate* signals

Some definitions:

- Column i produces a carry out by either *generating* a carry out or *propagating* a carry in to the carry out

- Calculate generate (G_i) and propagate (P_i) signals for each column:

  - **Generate:** Column i will generate a carry out if A_i **and** B_i are both 1.

$$G_i = A_i B_i$$

  - **Propagate:** Column i will propagate a carry in to the carry out if A_i **or** B_i is 1.

$$P_i = A_i + B_i$$

  - **Carry out:** The carry out of column i (C_i) is:

$$C_i = A_i B_i + (A_i + B_i)C_{i-1} = G_i + P_i C_{i-1}$$

Propagate and Generate Signals

Examples: Column propagate and generate signals:

Column propagate: $P_i = A_i + B_i$

Column generate: $G_i = A_i B_i$

$$
\begin{array}{rl}
1011 & A_{3:0} \\
+\ 0110 & B_{3:0} \\
\hline
\end{array}
$$

$$
\begin{array}{rl}
1011 & A_{3:0} \\
+\ 1001 & B_{3:0} \\
\hline
\end{array}
$$

Block Propagate and Generate

Now use column Propagate and Generate signals to compute **Block Propagate** and **Block Generate** signals for k-bit blocks, i.e.:

- Compute if a **k-bit group** will **propagate** a carry in (of the block) to the carry out (of the block)
- Compute if a **k-bit group** will **generate** a carry out (of the block)

Block Propagate and Generate

- **Example:** 4-bit blocks

  - **Block propagate signal:** $P_{3:0}$ (single-bit signal)

    - A carry-in would propagate through all 4 bits of the block:

      $$P_{3:0} = P_3 P_2 P_1 P_0$$

    - **Examples:**

```
    1011              1011
  + 0100            + 0001
  _____            _____
```

Block Propagate and Generate

- ## **Example:** 4-bit blocks

  - ### **Block propagate signal: $P_{3:0}$** (single-bit signal)
    - A carry-in would propagate through all 4 bits of the block:

      $$P_{3:0} = P_3 P_2 P_1 P_0$$

  - ### **Block generate signal: $G_{3:0}$** (single-bit signal)
    - A carry is generated:
      - in column 3, **or**
      - in column 2 and propagated through column 3, **or**
      - in column 1 and propagated through columns 2 and 3, **or**
      - in column 0 and propagated through columns 1-3

      $$G_{3:0} = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3$$
      $$G_{3:0} = G_3 + P_3 [G_2 + P_2 (G_1 + P_1 G_0)]$$

Block Propagate and Generate

- **Example:** 4-bit blocks

  - **Block generate signal:** $G_{3:0}$ (single-bit signal)

    - A carry is: generated in column 3, **or** generated in column 2 and propagated through column 3, **or** ...

$$G_{3:0} = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3$$

```
  1001              1110              0110
+ 1100            + 0100            + 0010
--------          --------          --------
```

Block Propagate and Generate

- **Example:** 4-bit blocks

  - **Block propagate signal: $P_{3:0}$** (single-bit signal)

    - A carry-in would propagate through all 4 bits of the block:

    $$P_{3:0} = P_3 P_2 P_1 P_0$$

  - **Block generate signal: $G_{3\text{-}0}$** (single-bit signal)

    - A carry is generated:
      - in column 3, **or**
      - in column 2 and propagated through column 3, **or**
      - in column 1 and propagated through columns 2 and 3, **or**
      - in column 0 and propagated through columns 1-3

    $$G_{3:0} = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3$$
    $$G_{3:0} = G_3 + P_3 [G_2 + P_2 (G_1 + P_1 G_0)]$$
    $$\boxed{C_3 = G_{3:0} + P_{3:0} C_{-1}}$$

Block Propagate and Generate

- **Example:** Block propagate and generate signals for 4-bit blocks ($P_{3:0}$ and $G_{3:0}$):

$$P_{3:0} = P_3 P_2 P_1 P_0$$

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$

$$C_3 = G_{3:0} + P_{3:0} C_{-1}$$

32-bit CLA with 4-bit Blocks
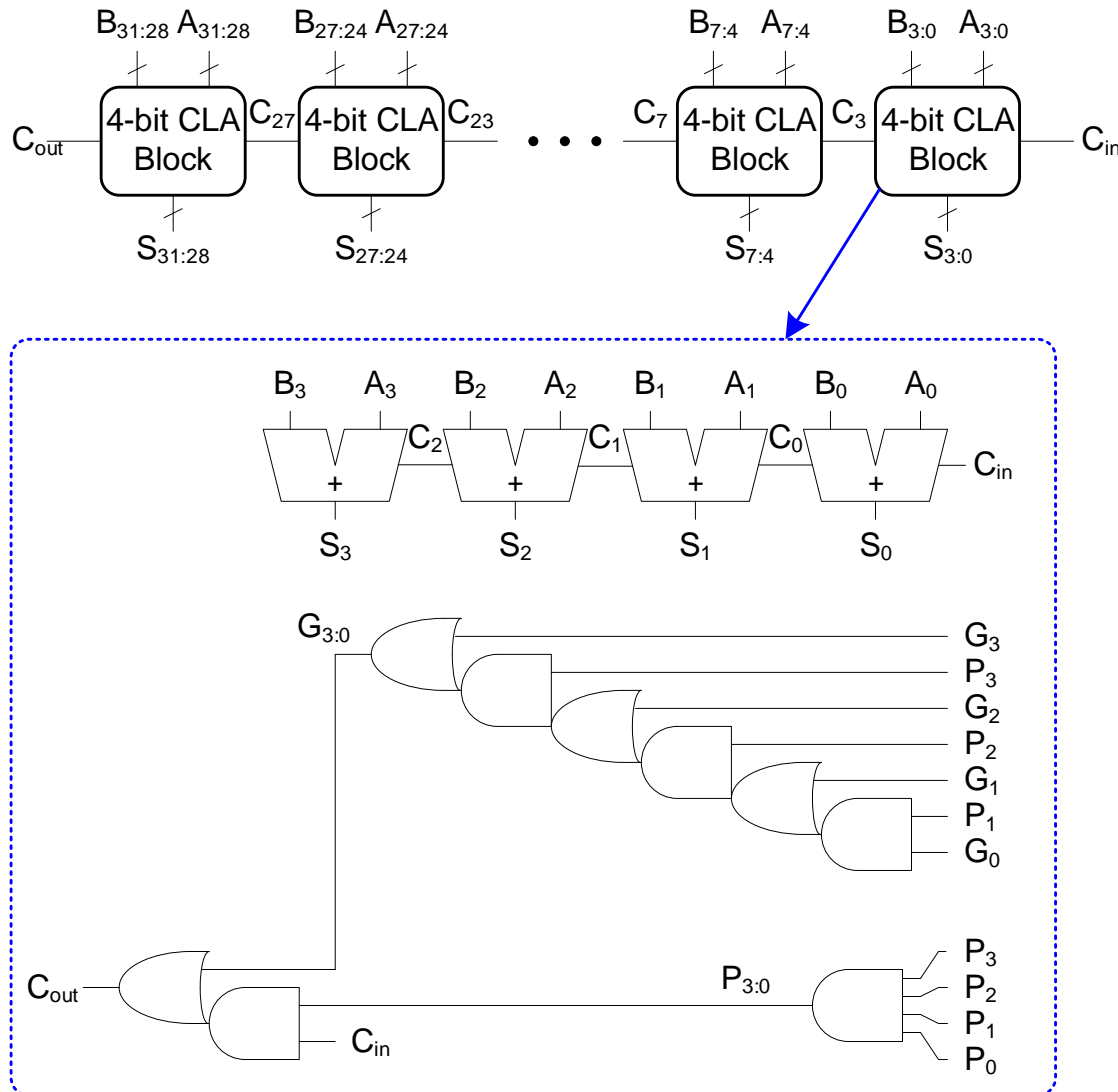
Digital Design & Computer Architecture   Digital Building Blocks
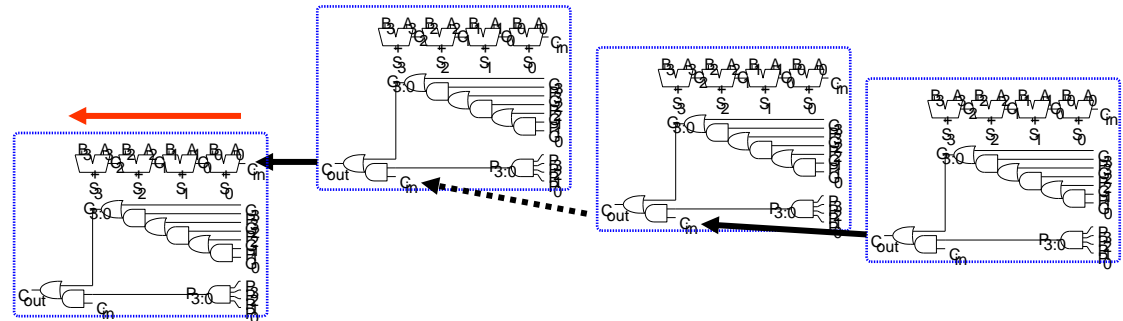
Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns

- **Step 2:** Compute G and P for k-bit blocks

- **Step 3:** C_{in} propagates through each k-bit propagate/generate logic (meanwhile computing sums)

- **Step 4:** Compute sum for most significant k-bit block

Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns

$$G_i = A_i B_i$$

$$P_i = A_i + B_i$$

Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k-bit blocks

$$P_{3:0} = P_3 P_2 P_1 P_0$$

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$

Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k-bit blocks
- **Step 3:** C_{in} propagates through each k-bit propagate/generate logic (meanwhile computing sums)
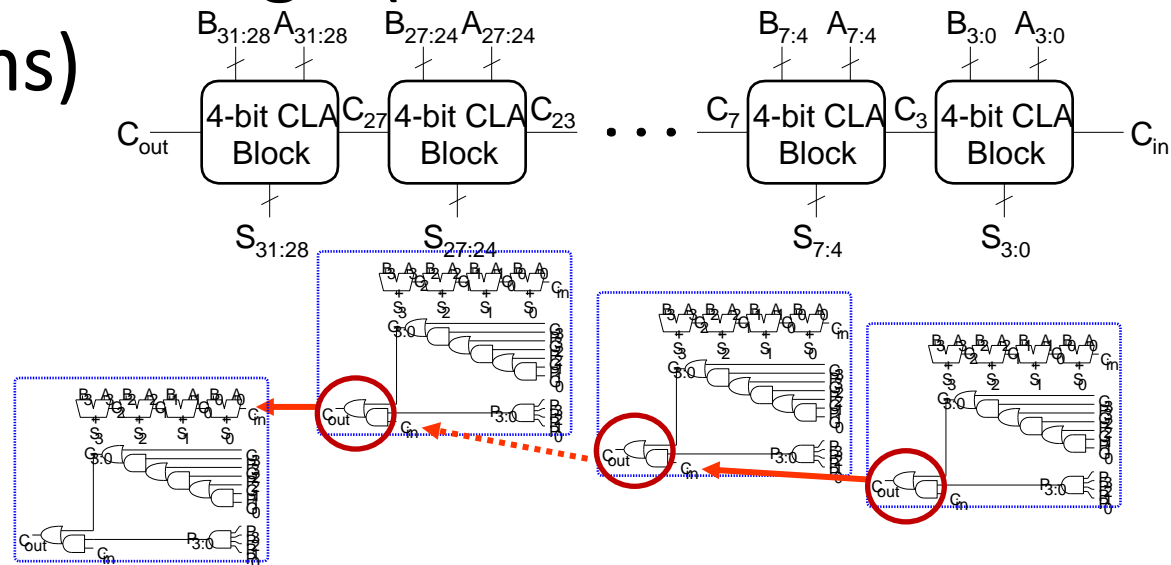
Carry-Lookahead Addition

- **Step 1:** Compute G_i and P_i for all columns

- **Step 2:** Compute G and P for k-bit blocks

- **Step 3:** C_{in} propagates through each k-bit propagate/generate logic (meanwhile computing sums)

- **Step 4:** Compute sum for most significant k-bit block

32-bit CLA with 4-bit Blocks
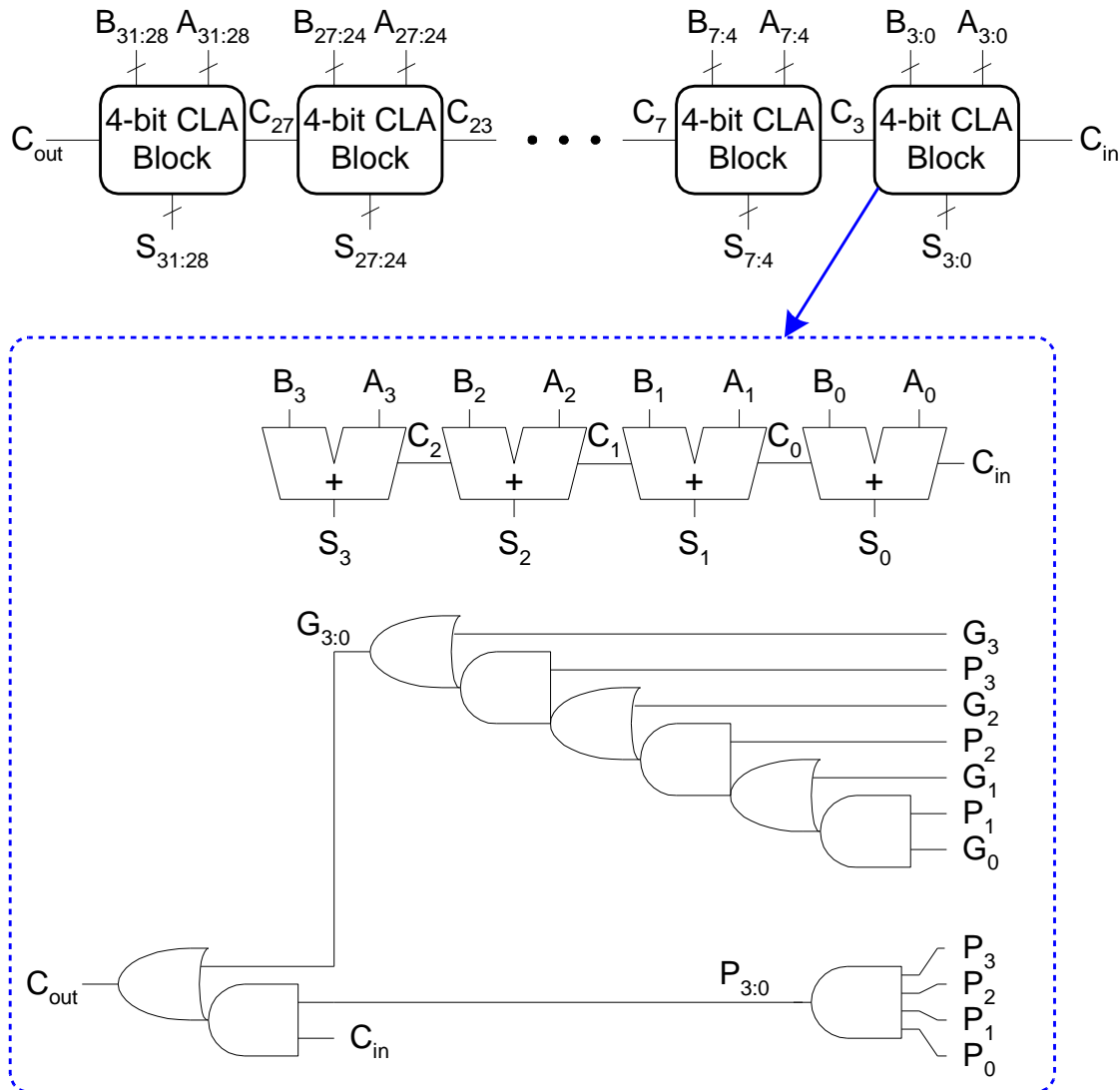
Carry-Lookahead Adder Delay

For *N*-bit CLA with *k*-bit blocks:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

- t_{pg} : delay to generate all P_i, G_i
- t_{pg_block} : delay to generate all $P_{i:j}$, $G_{i:j}$
- t_{AND_OR} : delay from C_{in} to C_{out} of final AND/OR gate in *k*-bit CLA block

An *N*-bit carry-lookahead adder is generally much faster than a ripple-carry adder for *N* > 16

Prefix Addition

Prefix Adder

- Computes carry in (C_{i-1}) for each column, then computes sum:

$$S_i = (A_i \oplus B_i) \oplus C_{i-1}$$

- It computes C_{i-1} by:
  - Computing G and P for 1-, 2-, 4-, 8-bit blocks, etc. until all G_i (carry in) known
  - $G_i = C_i$
- $\log_2 N$ stages

Prefix Adder

- Carry out either *generated* in a column or *propagated* from a previous column.

- Column -1 holds C_{in}, so

$$G_{-1} = C_{in}, P_{-1} = \text{X (not used)}$$

- Carry in to column *i* = carry out of column *i-1*:

$$C_{i-1} = G_{i-1:-1}$$

$G_{i-1:-1}$: generate signal spanning columns *i*-1 to -1

- Sum equation:

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

- **Goal:** Quickly compute $G_{0:-1}, G_{1:-1}, G_{2:-1}, G_{3:-1}, G_{4:-1}, G_{5:-1}, \ldots$ (called ***prefixes***)   (= C_0, C_1, C_2, C_3, C_4, C_5, …)

Prefix Adder

- Generate and propagate signals for a block spanning bits $i{:}j$

$$G_{i:j} = G_{i:k} + P_{i:k}\ G_{k\text{-}1:j}$$

$$P_{i:j} = P_{i:k}P_{k\text{-}1:j}$$

- In words:
  - **Generate:** block $i{:}j$ will generate a carry if:
    - **upper part** $(i{:}k)$ generates a carry $(G_{i:k})$ **or**
    - upper part $(i{:}k)$ propagates a carry $(P_{i:k})$ generated in lower part $(k\text{-}1{:}j)$ $(G_{k\text{-}1:j})$
  - **Propagate:** block $i{:}j$ will propagate a carry if *both* the upper and lower parts propagate the carry $(P_{i:k}\ AND\ P_{k\text{-}1:j})$

Prefix Adder Example

Step 1. Calculate P's and G's for **1-bit block**

Step 2. Calculate P's and G's for **2-bit blocks**

Step 3. Calculate P's and G's for **4-bit blocks**

Step 4. Continue to calculate P's and G's for larger blocks (8-bit, 16-bit, etc.)

Step 5. Use prefixes to calculate sums
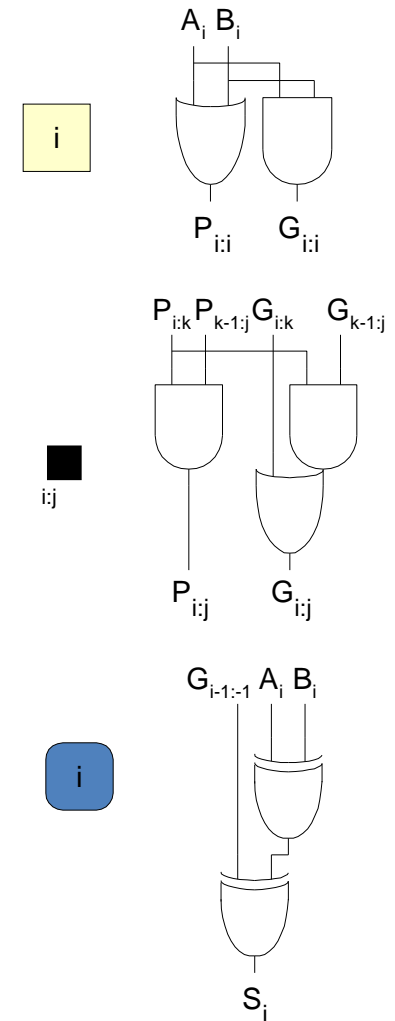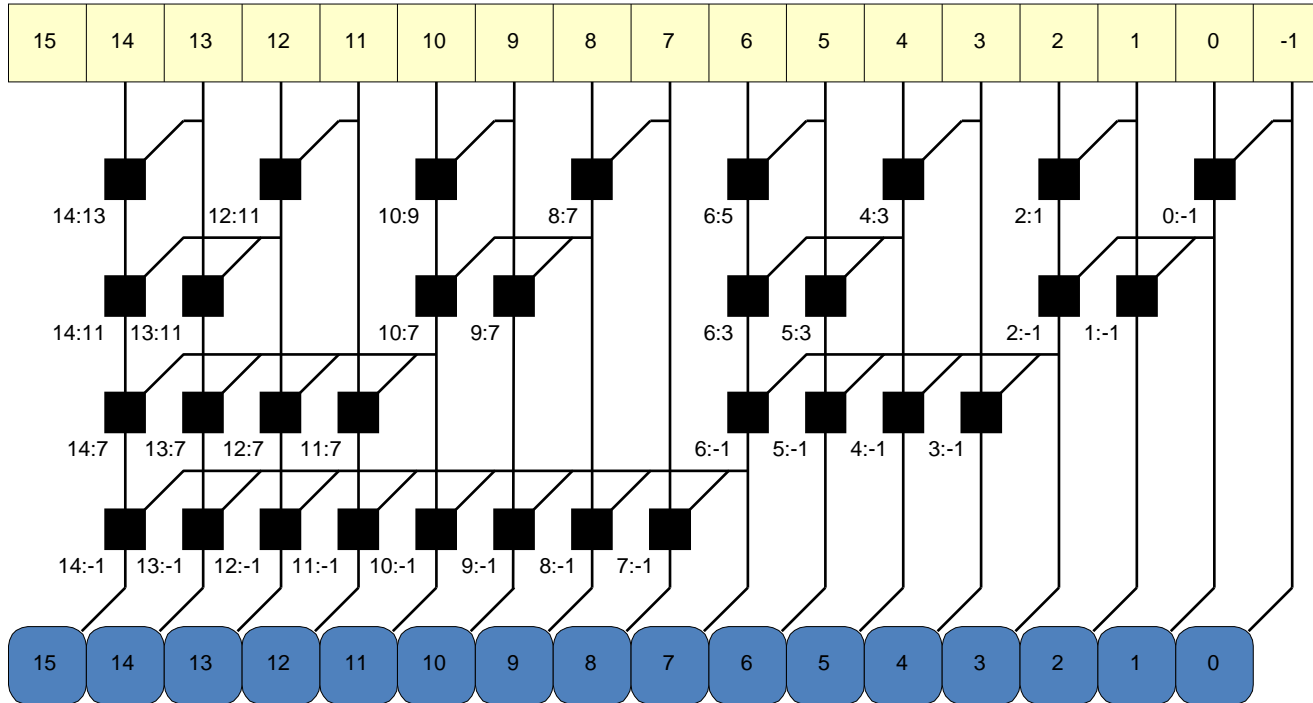
Prefix Adder Delay

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}}$$

t_{pg}: delay to produce P_i, G_i (AND or OR gate)

t_{pg_prefix}: delay of black prefix cell (AND-OR gate)

Adder Delay Comparisons

Compare the delay of: 32-bit ripple-carry, CLA, and prefix adders

- CLA has 4-bit blocks

- 2-input gate delay = 100 ps; full adder delay = 300 ps

$$t_{\mathbf{ripple}}$$

$$t_{CLA}$$

$$t_{PA}$$

Subtracters & Comparators

Subtracter

$$A - B = A + \overline{B} + 1$$

Symbol   **Implementation**
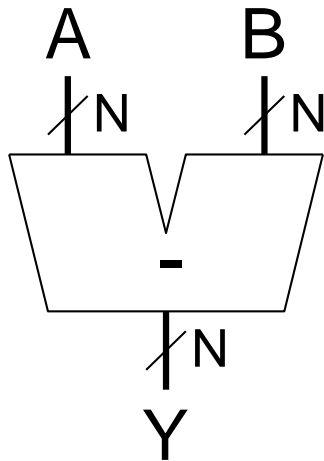
Symbol

Implementation

A   B

/4   /4

=

Equal
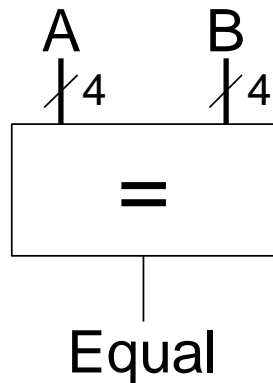
A < B if A-B is negative

Beware of overflow

ALU:

Arithmetic Logic Unit

ALU: Arithmetic Logic Unit

ALU should perform:

- Addition

- Subtraction

- AND

- OR

ALU: Arithmetic Logic Unit

ALUControl$_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: **Perform *A* OR *B***

ALUControl$_{1:0}$ = 11
Result = A* OR *B
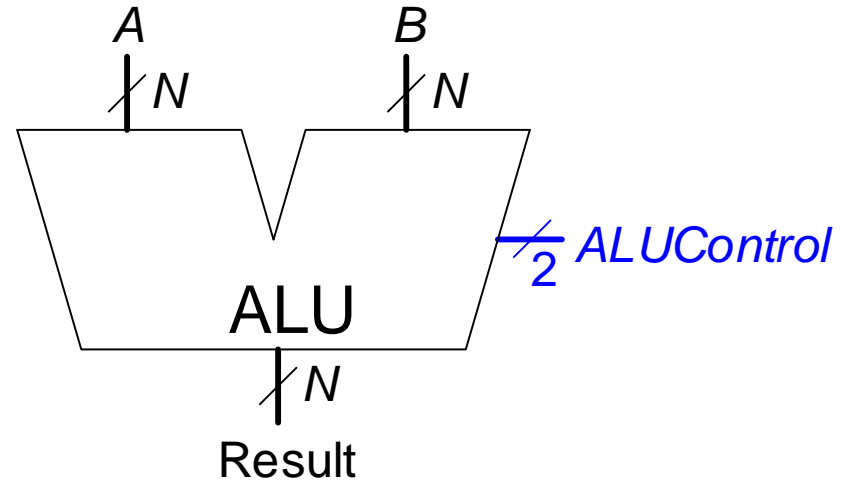
ALU: Arithmetic Logic Unit

ALUControl$_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: Perform *A* OR *B*

ALUControl$_{1:0}$ = 11

Mux selects output of OR gate as *Result,* so:

Result = A OR B
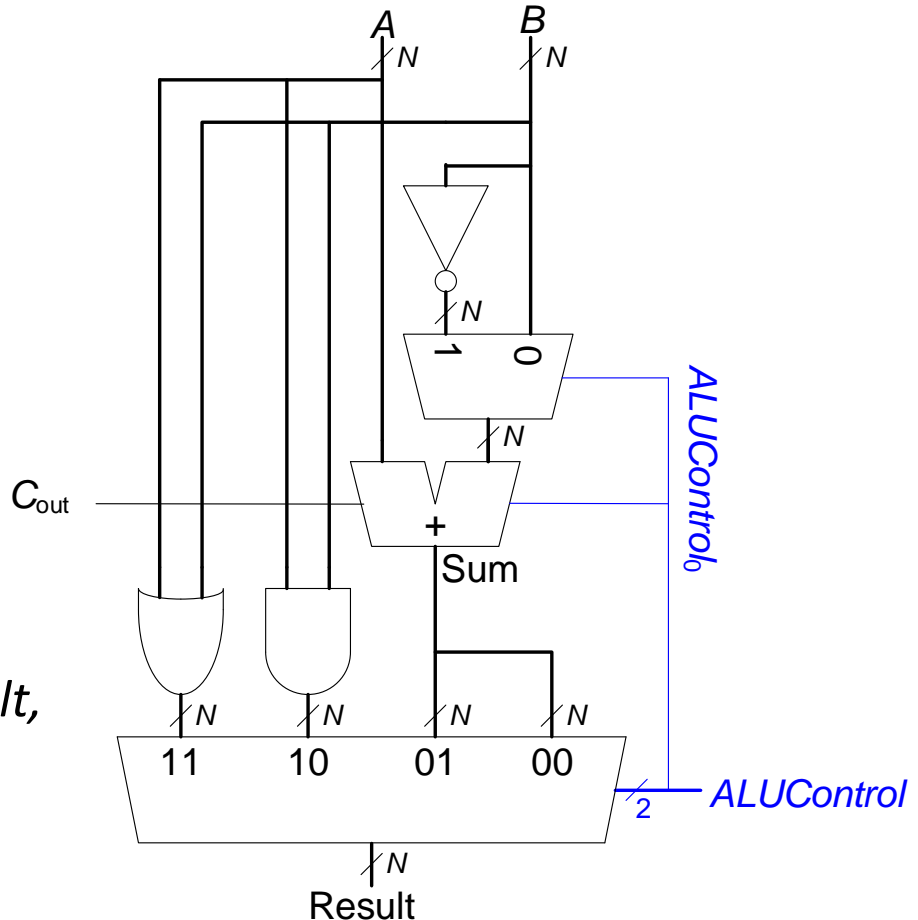
Digital Design & Computer Architecture   Digital Building Blocks

ALU: Arithmetic Logic Unit

ALUControl$_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: Perform $A + B$

$ALUControl_{1:0}$ = 00

$ALUControl_0$ = 0, so:

   C_{in} to adder = 0

   2nd input to adder is B

Mux selects Sum as $Result,$ so

   $Result = A + B$
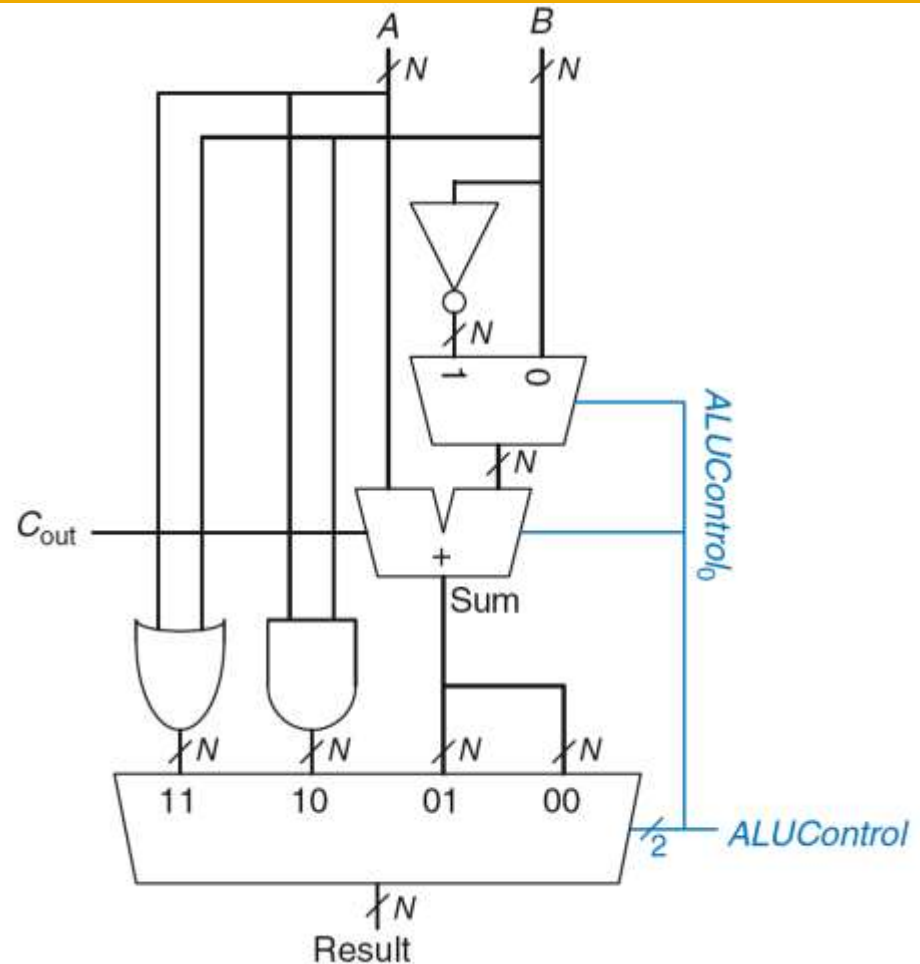
ALU with Status Flags

Flag	Description
N	*Result* is **N**egative
Z	*Result* is **Z**ero
C	Adder produces **C**arry out
V	Adder o**V**erflowed
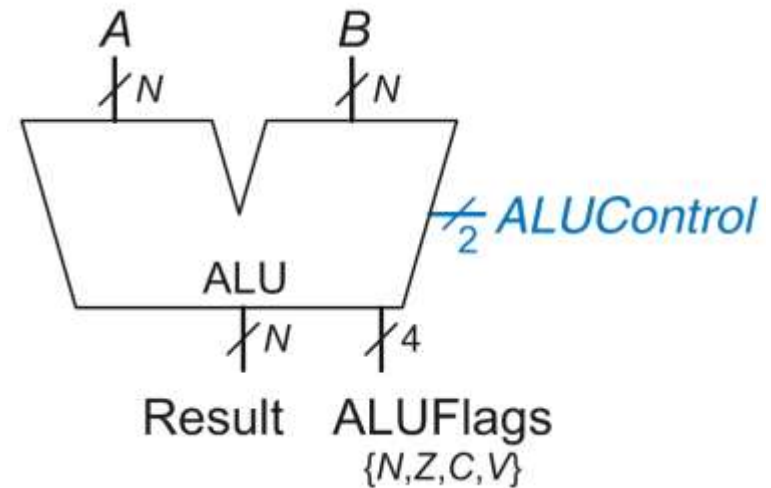
ALU with Status Flags
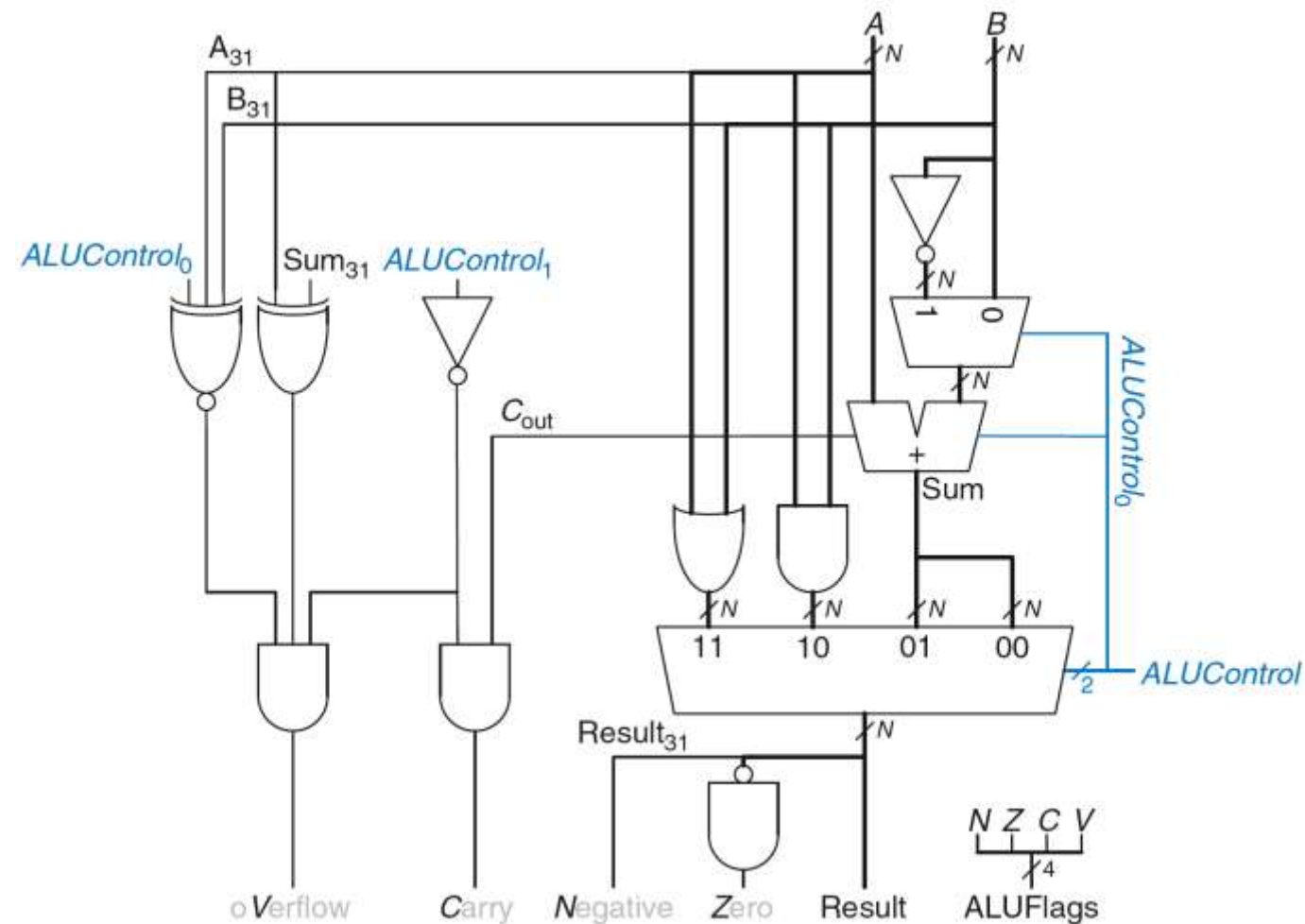
ALU with Status Flags: **N**egative
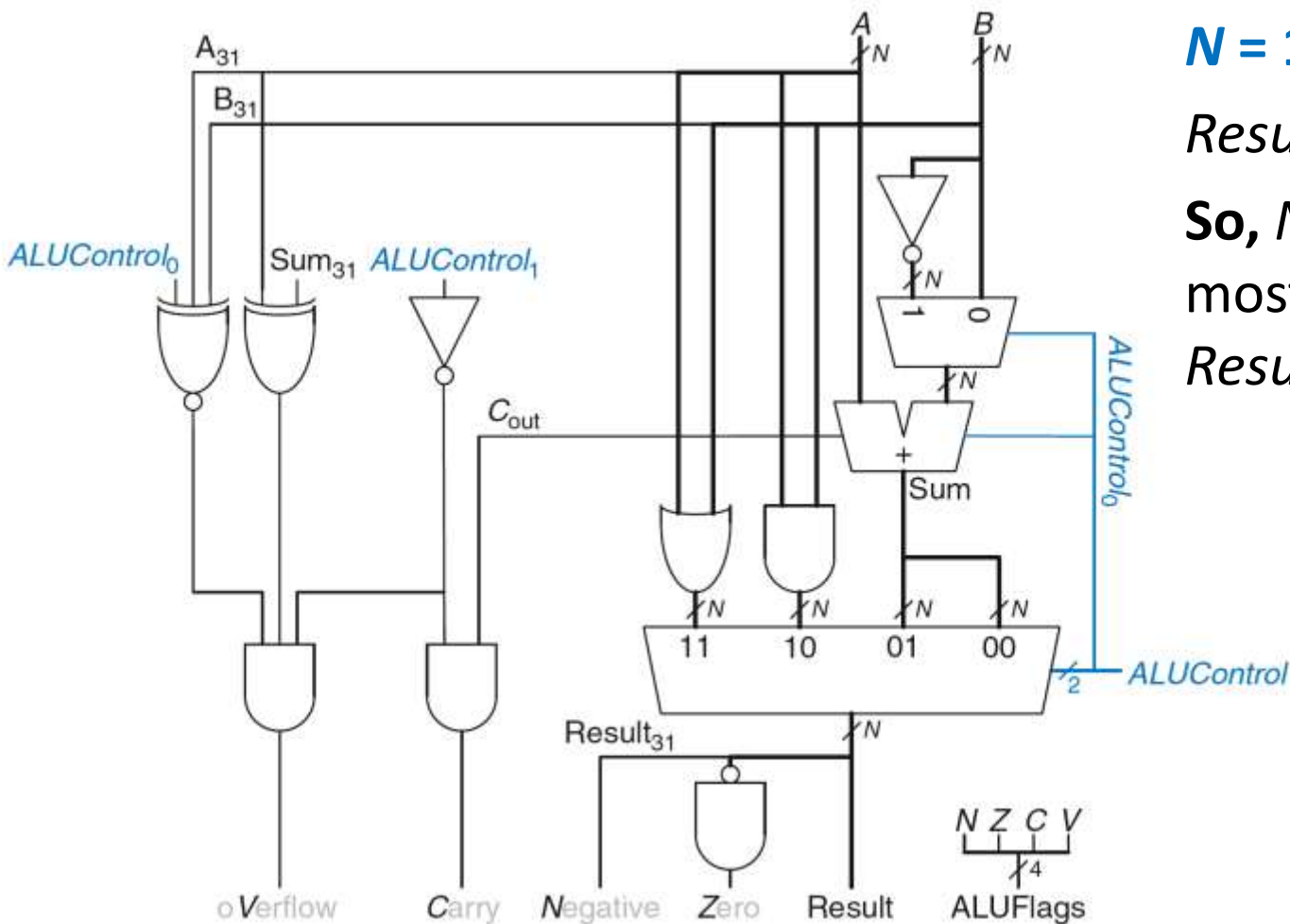
N = 1 if:

Result is **negative**

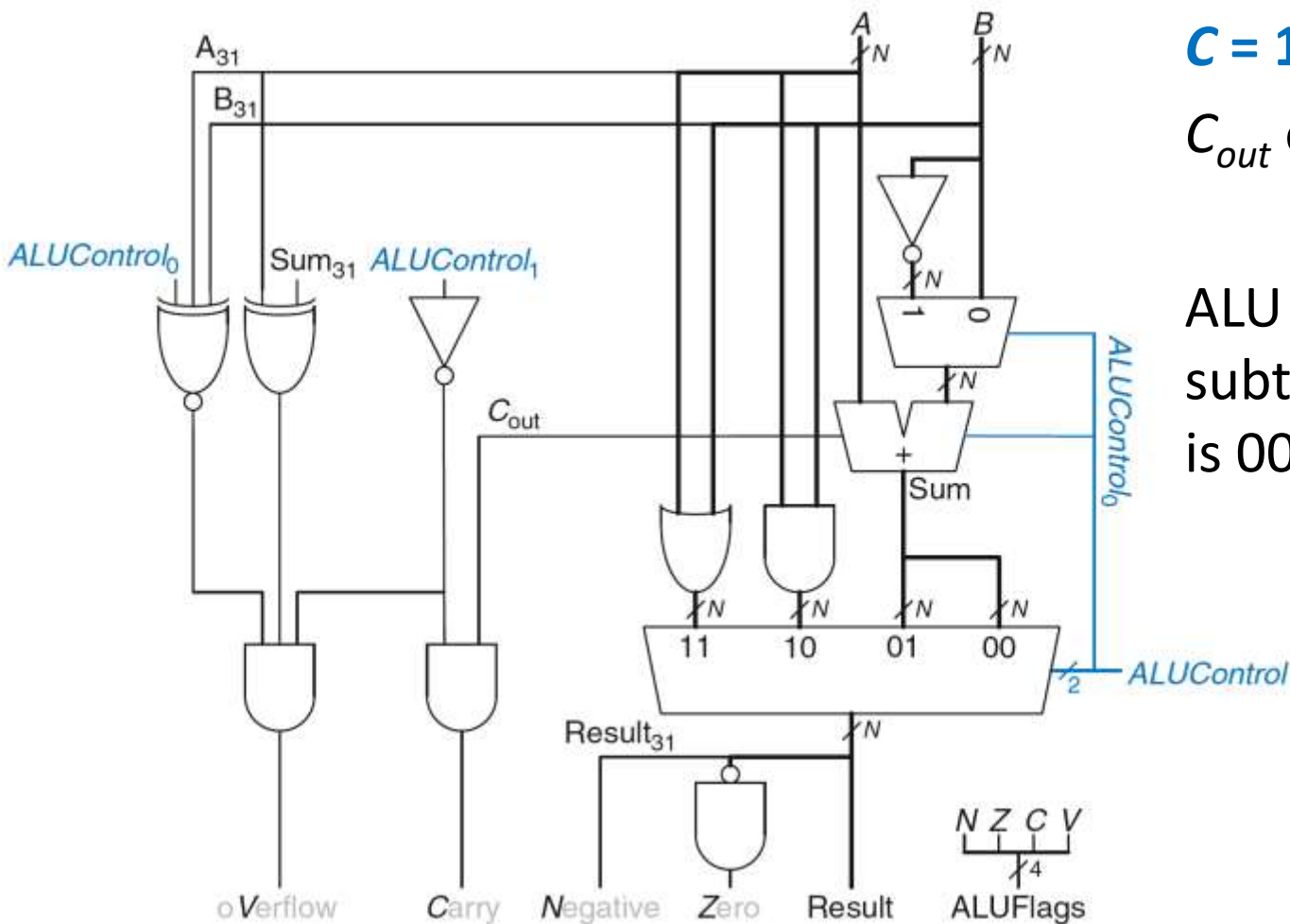**So,** *N* is connected to most significant bit of *Result.*

ALU with Status Flags: Zero

Z = 1 if:

all of the bits of *Result* are 0
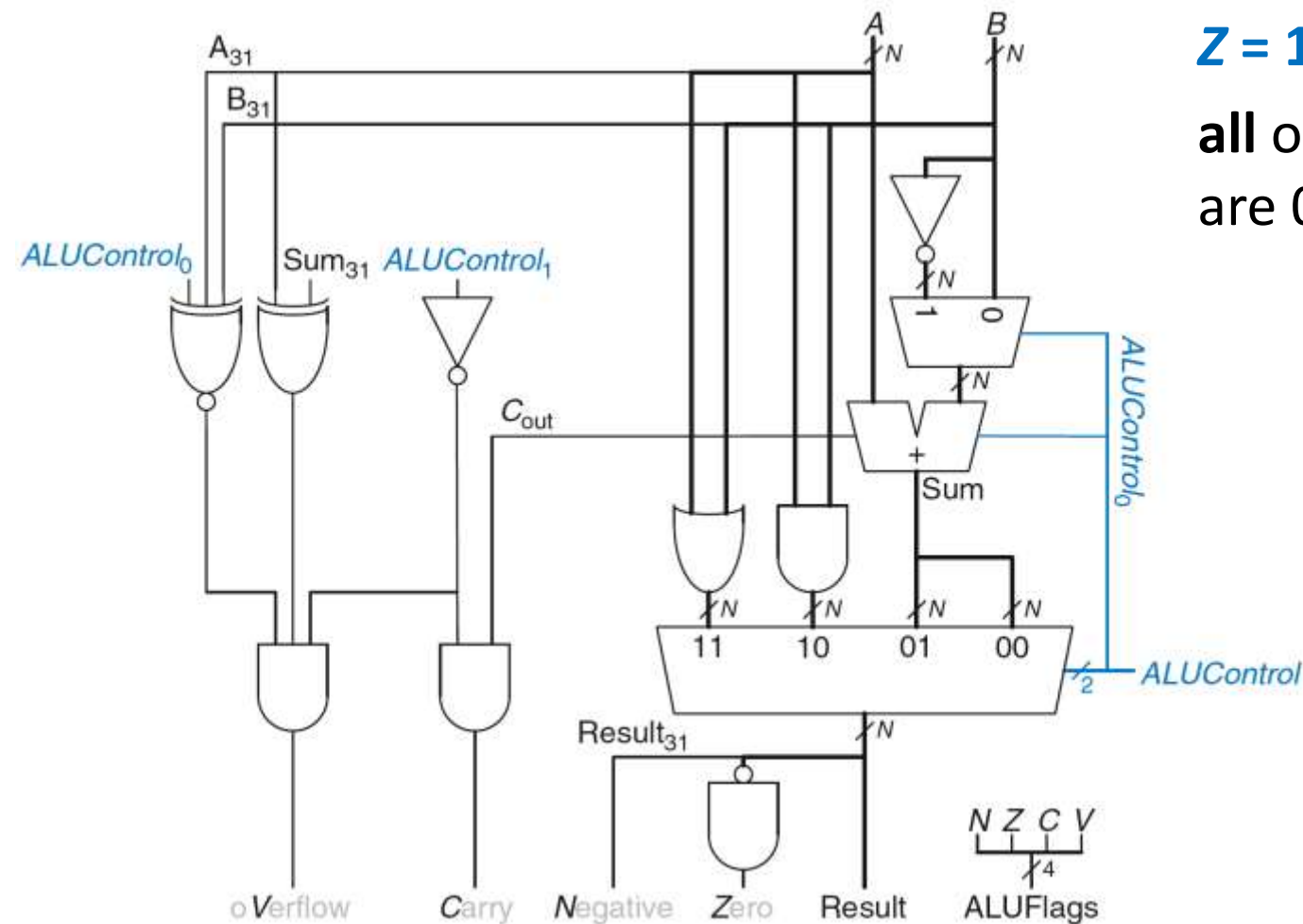
ALU with Status Flags: Carry

C = 1 if:

C_{out} of Adder is 1
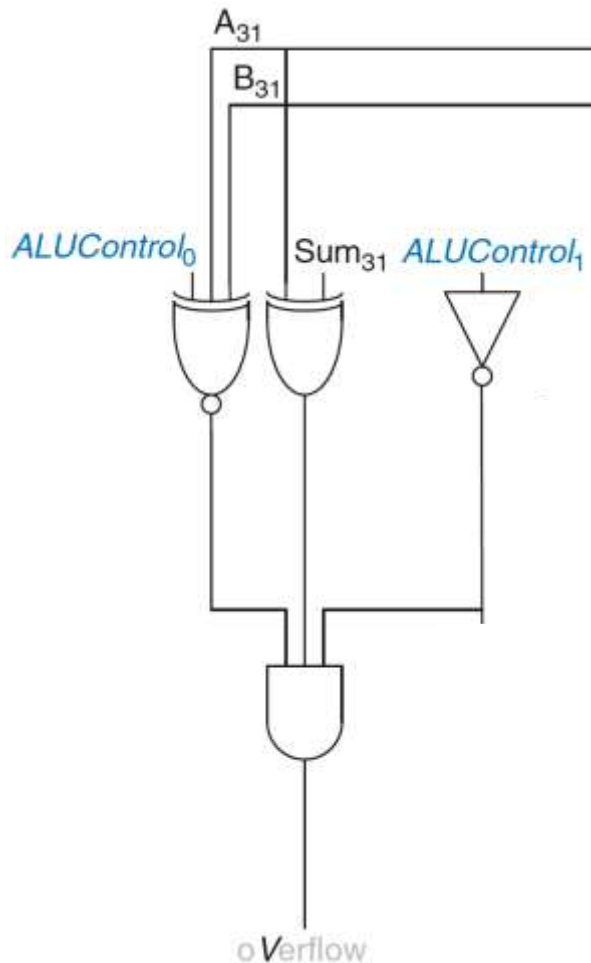
AND

ALU is adding or subtracting (ALUControl is 00 or 01)

ALU with Status Flags: oVerflow

V = 1 if:

The addition of 2 **same-signed numbers** produces a result with the **opposite sign**
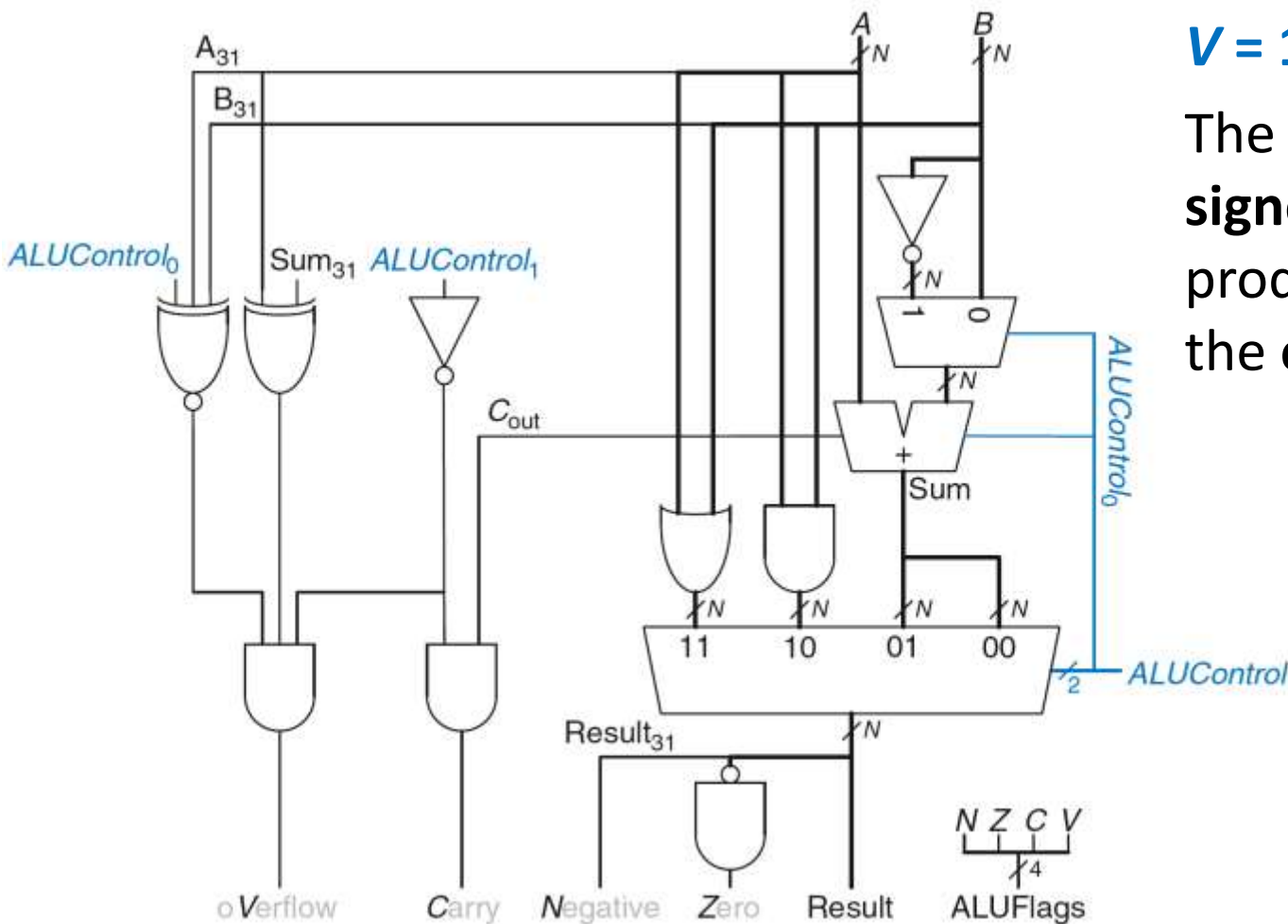
ALU with Status Flags: oVerflow

$V = 1$ if:

ALU is performing addition or subtraction ($ALUControl_1 = 0$)

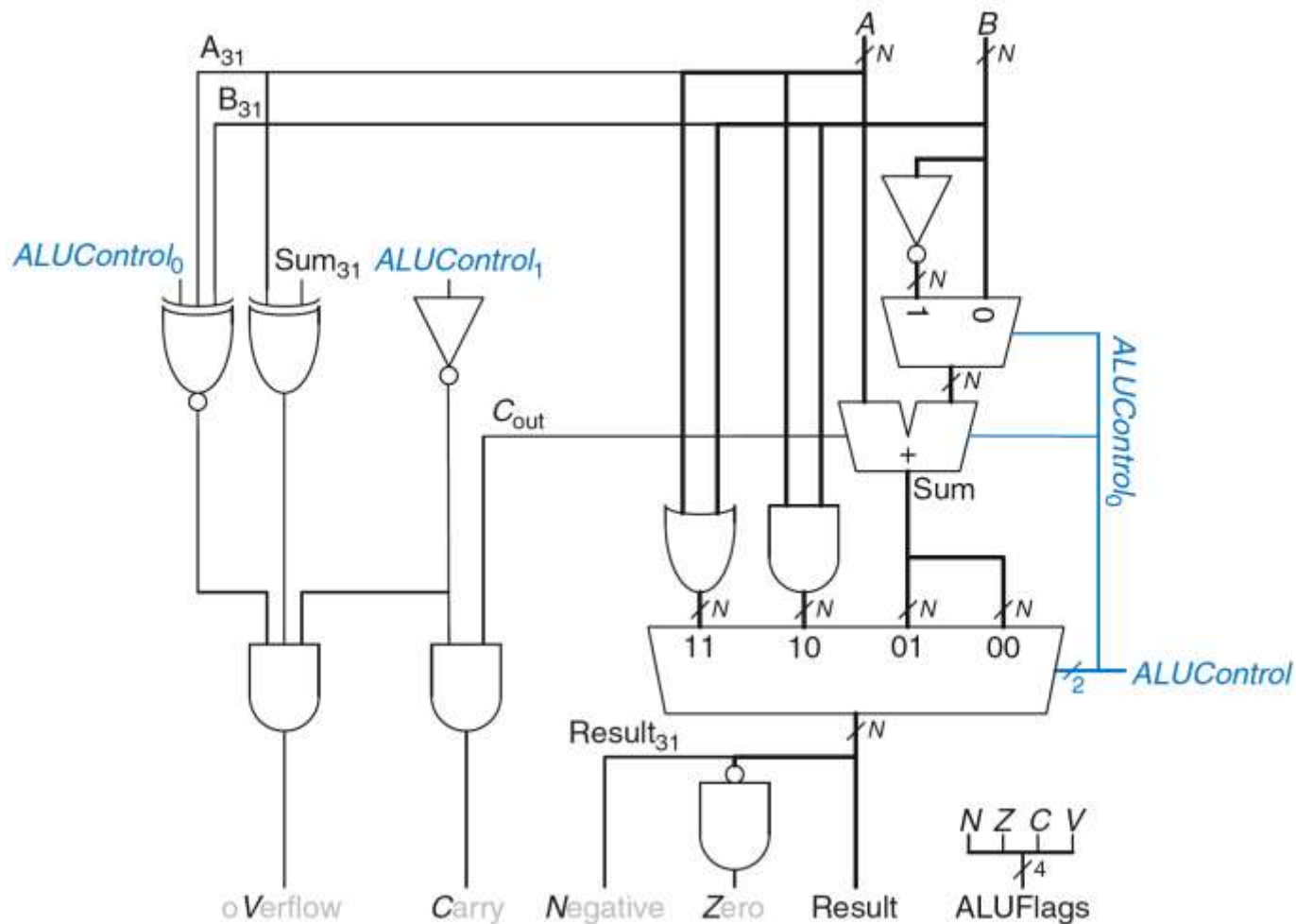**AND**

A and Sum have opposite signs

AND

A and B have same signs for addition ($ALUControl_0 = 0$)   **OR**

A and B have different signs for subtraction ($ALUControl_0 = 1$)

ALU with Status Flags

Comparison based on Flags

Compare by subtracting and checking flags
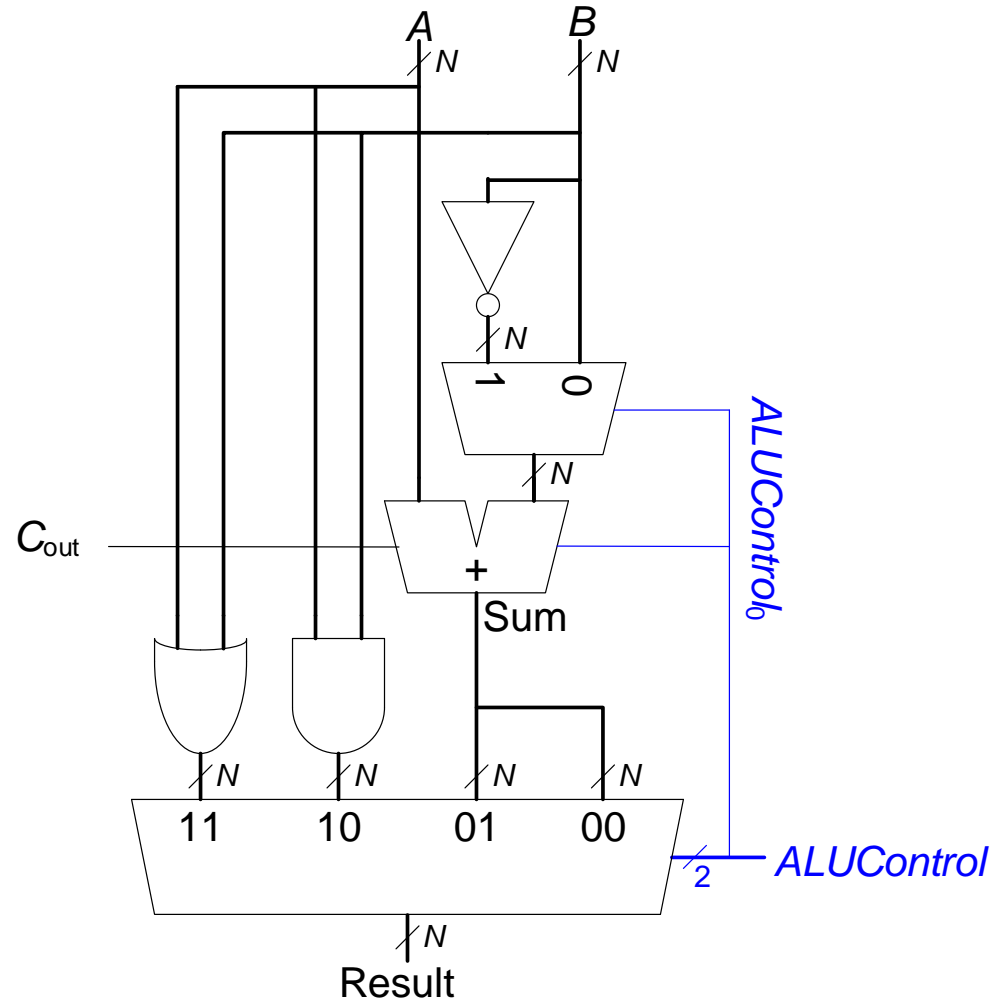
Different for signed and unsigned

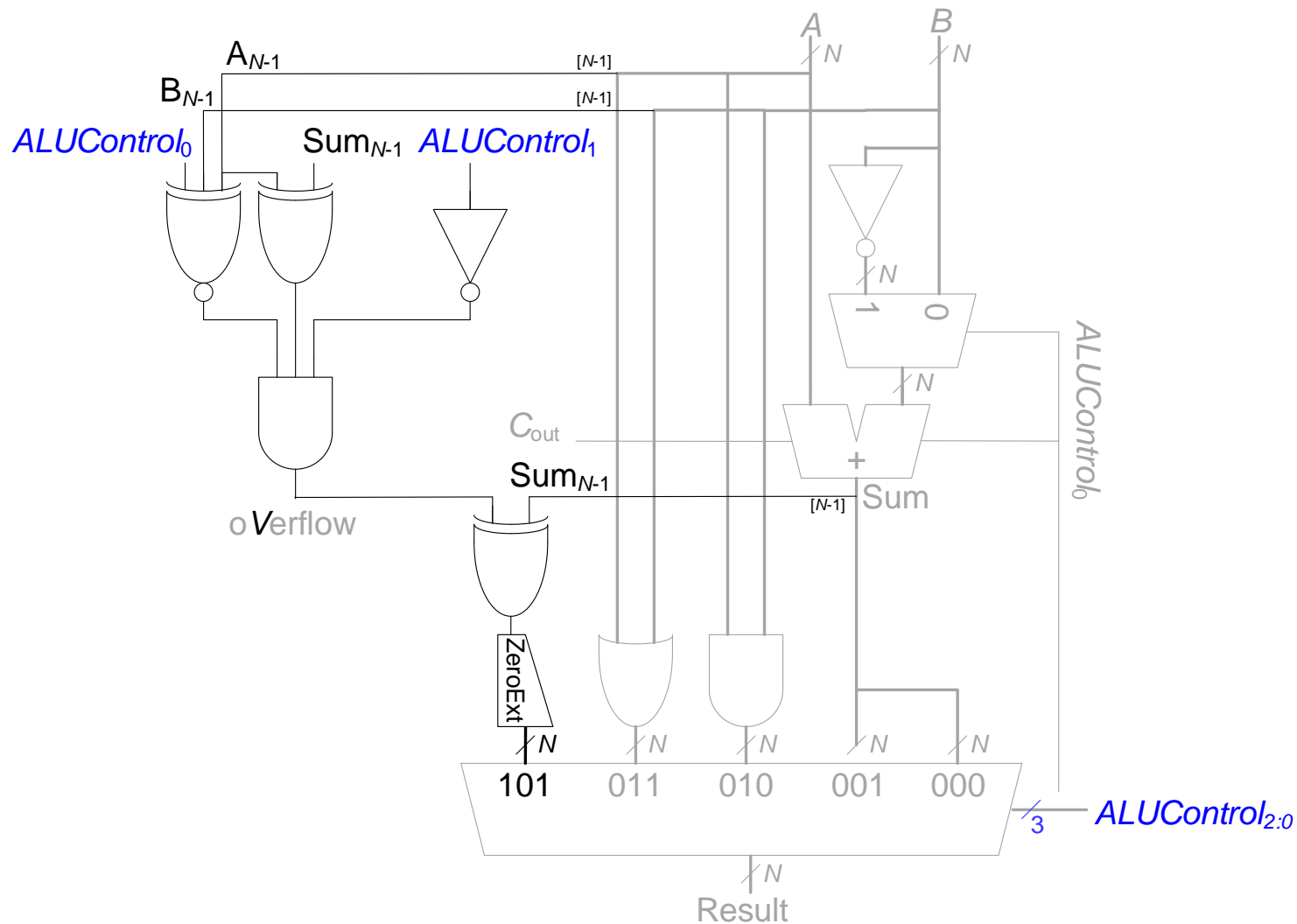| Comparison	Signed	Unsigned
==	Z	Z
!=	~Z	~Z
<	N ^ V	~C
<=	Z | (N ^ V)	Z | ~C
>	~Z & ~(N ^ V)	~Z & C
>=	~(N ^ V)	C

Other ALU Operations

- **Set Less Than** (also called Set if Less Than)
  - **Sets lsb** of result **if A < B**
    - Result = 0000...001 if A < B
    - Result = 0000...000 otherwise
  - Comes in signed and unsigned flavors
- **XOR**
  - Result = A XOR B

Extending ALU: SLT

ALUControl$_{2:0}$	Function
00	add
01	subtract
10	and
11	or
	SLT

Shifters, Multipliers, & Dividers

Shifters

Logical shifter: shifts value to left or right and fills empty spaces with 0's
- Ex: 11001 >> 2 =
- Ex: 11001 << 2 =

Arithmetic shifter: same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb)
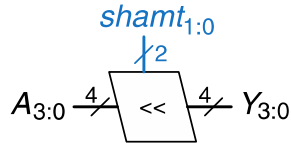- Ex: 11001 >>> 2 =
- Ex: 11001 <<< 2 =

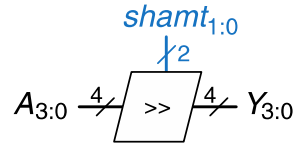**Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
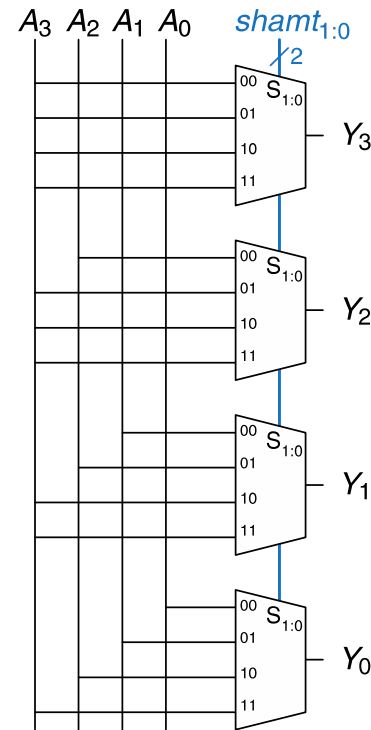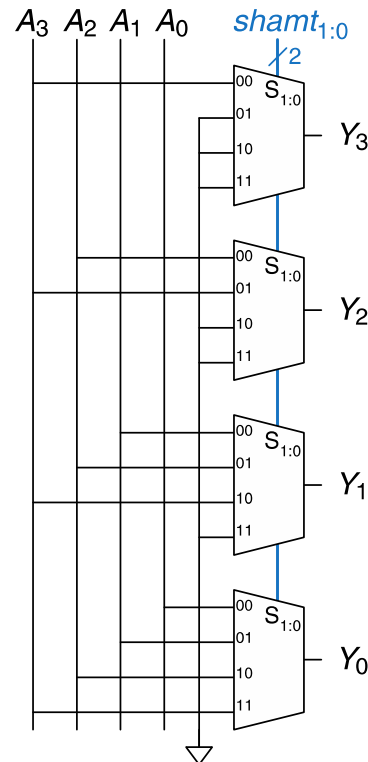- Ex: 11001 ROR 2 =
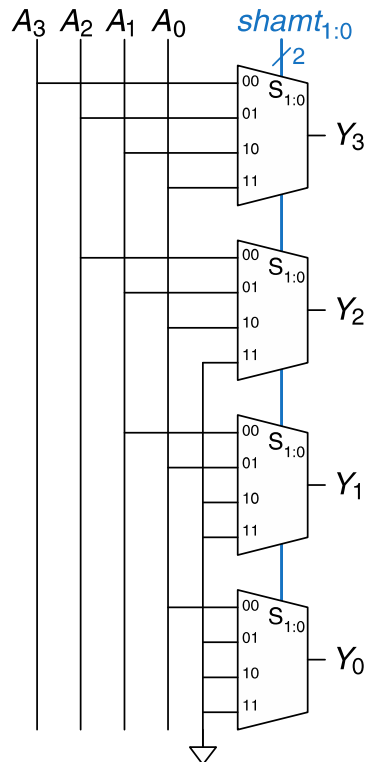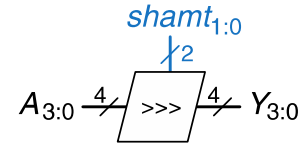- Ex: 11001 ROL 2 =

Shifter Design

Shift Left

Logical Shift Right

Arithmetic Shift Right

Shifters as Multipliers and Dividers

- $A \ll N = A \times 2^N$

- $A \ggg N = A \div 2^N$

Multipliers

- **Partial products** formed by multiplying a single digit of the multiplier with multiplicand
- **Shifted** partial products **summed** to form result

Decimal   Binary

```
        230    multiplicand      0101
  x      42    multiplier      x 0111
  --------                     --------
       460     partial           0101
  + 920        products          0101
  --------                       0101
      9660                    + 0000
                             ----------
               result        0100011
```

230 x 42 = 9660   5 x 7 = 35

		A_3	A_2	A_1	A_0
x		B_3	B_2	B_1	B_0

A_3B_0	A_2B_0	A_1B_0	A_0B_0
A_3B_1	A_2B_1	A_1B_1	A_0B_1
A_3B_2	A_2B_2	A_1B_2	A_0B_2
A_3B_3	A_2B_3	A_1B_3	A_0B_3

| P_7 | P_6 | P_5 | P_4 | P_3 | P_2 | P_1 | P_0 |
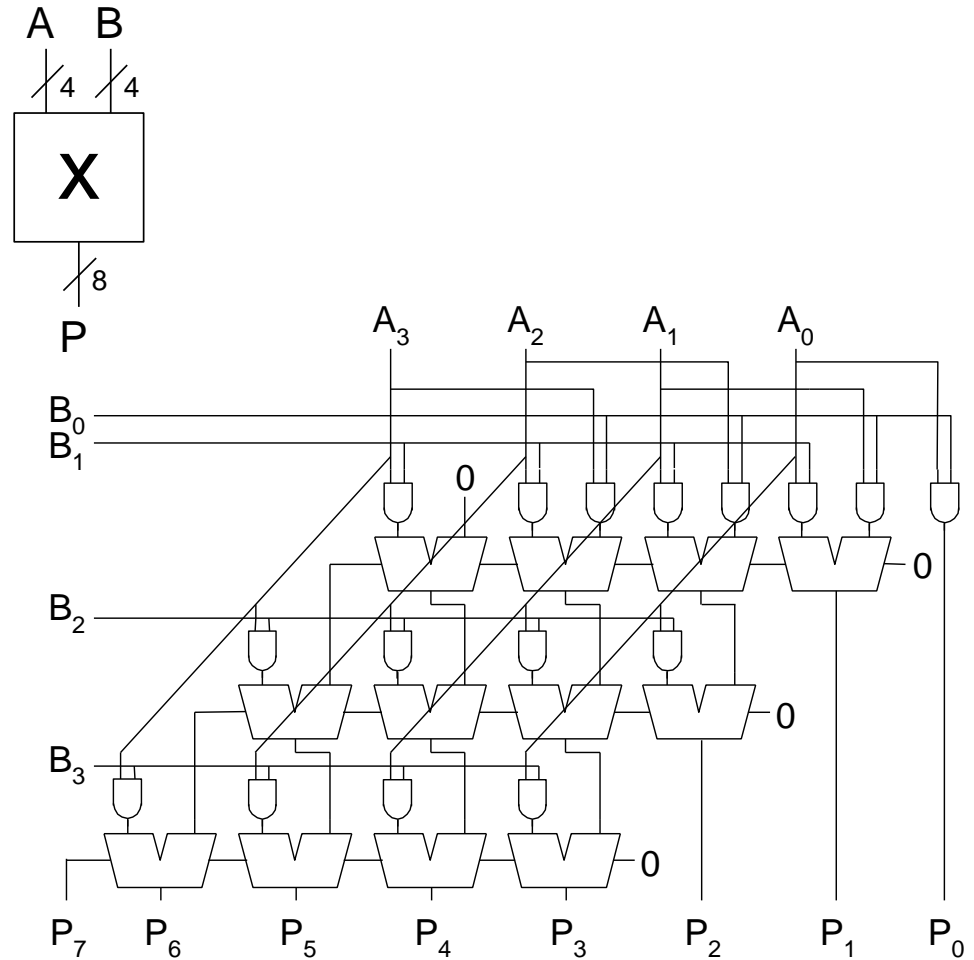
Dividers

$$A/B = Q + R/B$$

Decimal Example: 2584/15 = 172 R4

Long-Hand:

$$
\begin{array}{r}
172 \ \text{R4} \\
15\overline{)2584} \\
-15 \\
\hline
108 \\
-105 \\
\hline
34 \\
-30 \\
\hline
4
\end{array}
$$

Long-Hand Revisited:

$$
\begin{array}{r}
0002 \\
- \ \ 15 \\
\hline
-13
\end{array}
\qquad
\frac{0}{\scriptstyle 3 \ 2 \ 1 \ 0}
$$

$$
\begin{array}{r}
0025 \\
- \ \ 15 \\
\hline
10
\end{array}
\qquad
\frac{0 \ 1}{\scriptstyle 3 \ 2 \ 1 \ 0}
$$

$$
\begin{array}{r}
0108 \\
- \ 105 \\
\hline
3
\end{array}
\qquad
\frac{0 \ 1 \ 7}{\scriptstyle 3 \ 2 \ 1 \ 0}
$$

$$
\begin{array}{r}
0034 \\
- \ \ 30 \\
\hline
4
\end{array}
\qquad
\frac{0 \ 1 \ 7 \ 2}{\scriptstyle 3 \ 2 \ 1 \ 0}
$$

Dividers

$$A/B = Q + R/B$$

Decimal: 2584/15 = 172 R4   **Binary:**   1101/0010 = 0110 R1

```
  0002                            0001
-   15      0 _ _ _             - 0010      0 _ _ _
  -13       3 2 1 0               1111      3 2 1 0

  0025                            0011
-   15      0 1 _ _             - 0010      0 1 _ _
   10       3 2 1 0               0001      3 2 1 0

  0108                            0010
-  105      0 1 7 _             - 0010      0 1 1 _
    3       3 2 1 0               0000      3 2 1 0

  0034                            0001
-   30      0 1 7 2             - 0010      0 1 1 0  R1
    4       3 2 1 0               1111      3 2 1 0
```

Dividers

$A/B = Q + R/B$

Binary: 1101/10 = 0110 R1

$R' = 0$

for i = N-1 to 0

   $R = \{R' \ll 1, A_i\}$

   $D = R - B$

   if $D < 0$, $Q_i = 0$; $R' = R$

   else $Q_i = 1$; $R' = D$

$R = R'$

```
  0001
- 0010        0
  1111        3 2 1 0
```

```
  0011
- 0010        0 1
  0001        3 2 1 0
```

```
  0010
- 0010        0 1 1
  0000        3 2 1 0
```

```
  0001
- 0010        0 1 1 0  R1
  1111        3 2 1 0
```

4 x 4 Divider

Legend

Division: $A/B = Q + R/B$

$R' = 0$

for $i = N\text{-}1$ to 0

  $R = \{R' << 1, A_i\}$

  $D = R - B$

  if $D < 0$, $Q_i = 0$, $R' = R$

  else $Q_i = 1$, $R' = D$

$R = R'$

Each row computes one iteration of the division algorithm.

4 x 4 Divider

Each row computes one iteration of the division algorithm.

Fixed-Point Numbers

Number Systems

Numbers we can represent using binary representations

- **Positive numbers**
  - Unsigned binary
- **Negative numbers**
  - Two's complement
  - Sign/magnitude numbers

What about **fractions**?

Numbers with Fractions

Two common notations:

- **Fixed-point:** binary point fixed

- **Floating-point:** binary point floats to the right of the most significant 1

Fixed-Point Numbers

- 6.75 using 4 integer bits and 4 fraction bits:

$$01101100$$

$$0110.1100$$

$$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

- Binary point is implied
- The number of integer and fraction bits must be agreed upon beforehand

Unsigned Fixed Point Formats

- **Ua.b:** unsigned number with
  - **a** integer bits
  - **b** fractional bits.
- **Example: 6.75** is
  - **U4.4:** 01101100
  - **U3.5:** 11011000
  - **U6.2:** 00011011
- 8, 16, and 32-bit fixed point numbers are common
  - **U8.8** often represents sensor data, audio, pixels
  - **U16.16** used for higher precision signal processing

Signed Fixed Point Formats

- **Qa.b:** signed 2's complement number with
  - **a** integer bits (including the sign bit)
  - **b** fractional bits

- **To negate a Q fixed point number:**
  - Invert the bits
  - Add one to the LSB

- **Example:** write **-6.75** in Q4.4
  - 6.75 =   01101100
  - Invert: 10010011
  - Add 1 LSB: 10010100

- **Q1.15** (aka Q15) is common for signal processing (1, -1]

Saturating Arithmetic

- Fixed point **overflow** is usually bad
  - Produces undesired artifacts:
    - Video: dark pixel in middle of bright pixels
    - Audio: clicking sounds
- **Saturating arithmetic**
  - Instead of overflowing, use largest value
  - In U4.4: 11000000 + 01111000 = 11111111

         12   + 7.5   = 15.9375

Floating-Point Numbers

Floating-Point Numbers

- Binary point floats to the right of the most significant 1
- Similar to decimal scientific notation

- For example, write 273_{10} in scientific notation:

$$273 = 2.73 \times 10^2$$

- In general, a number is written in scientific notation as:

$$\pm M \times B^E$$

  - M = mantissa
  - B = base
  - E = exponent
  - In the example, M = 2.73, B = 10, and E = 2

Floating vs. Fixed Point Numbers

- Floating point numbers are like **scientific notation**
  - Allow a **greater dynamic range** of smallest to largest
  - **Arithmetic is harder**
    - Mantissa must be aligned before adding
    - This costs performance and power
- Fixed point numbers are **harder for the programmer**
  - **Smaller dynamic range**
  - Take care of **overflow**

- **Floating Point** is preferred for general-purpose computing where programming time is most important
- **Fixed Point** is preferred for signal processing performance, power, and hardware cost matter most
  - Machine learning, video

Floating-Point Numbers

1 bit	8 bits	23 bits

Sign   **Exponent**   **Mantissa**
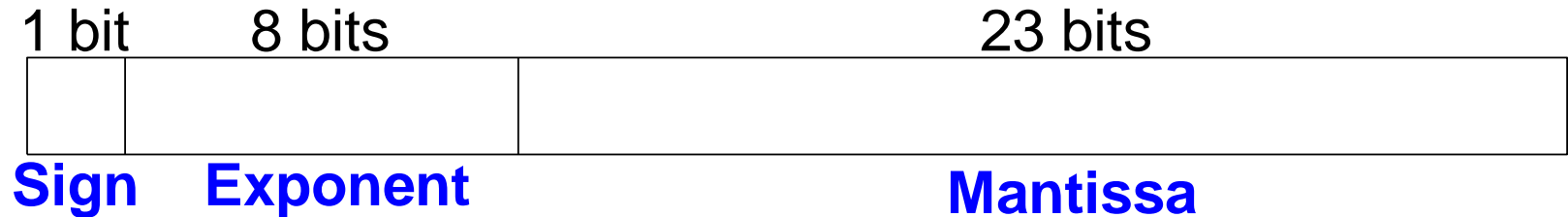
- **Example:** represent the value 228_{10} using a 32-bit floating point representation

We show three versions – **the final version** is called the:

IEEE 754 floating-point standard

Floating-Point Representation 1
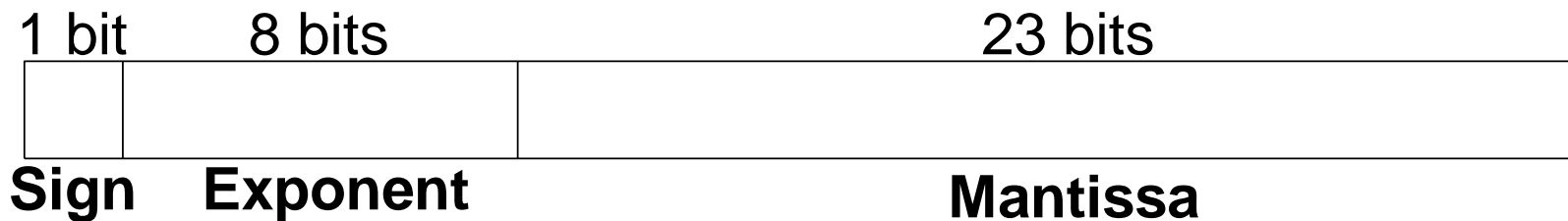
1. Convert decimal to binary:

$$228_{10} = 11100100_2$$

2. Write the number in "binary scientific notation":

$$11100100_2 = 1.11001_2 \times 2^7$$

3. Fill in each field of the 32-bit floating point number:

   – The sign bit is positive (0)

   – The 8 exponent bits represent the value 7

   – The remaining 23 bits are the mantissa

1 bit	8 bits	23 bits
Sign	**Exponent**	**Mantissa**

Floating-Point Representation 2

- First bit of the mantissa is always 1:

  - $228_{10} = 11100100_2 = \mathbf{1}.\mathbf{11001} \times 2^7$

- So, no need to store it: *implicit leading 1*

- Store just **fraction bits** in 23-bit field

| 1 bit	8 bits	23 bits
0	00000111	110 0100 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**
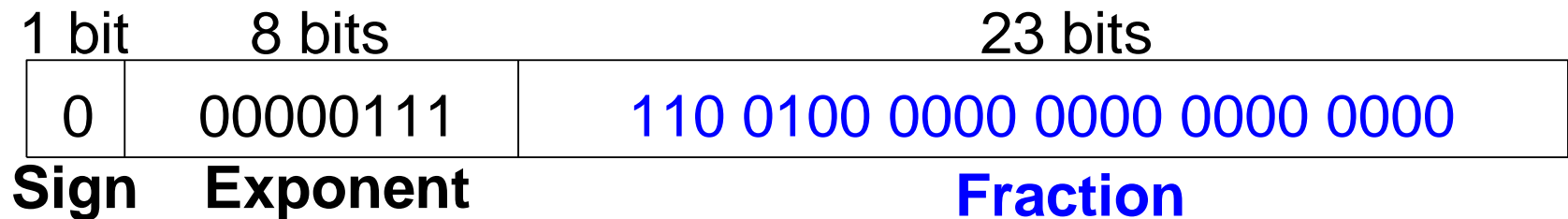
Floating-Point Representation 3

- **Biased exponent:** bias = 127 (01111111_2)

  – Biased exponent = bias + exponent

  – Exponent of 7 is stored as:

$$127 + 7 = 134 = 0x10000110_2$$

- The **IEEE 754 32-bit floating-point representation** of 228_{10}

1 bit	8 bits	23 bits
0	10000110	110 0100 0000 0000 0000 0000
Sign	**Biased Exponent**	**Fraction**
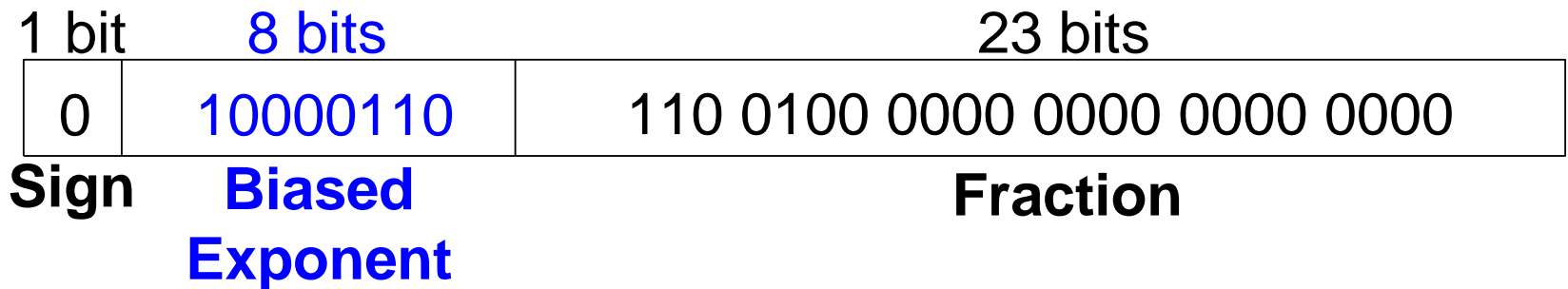
in hexadecimal: **0x43640000**

Floating-Point Example

Write **-58.25$_{10}$** in floating point (IEEE 754)

1. Convert magnitude of decimal to binary:

    58.25$_{10}$ = 111010.01$_2$
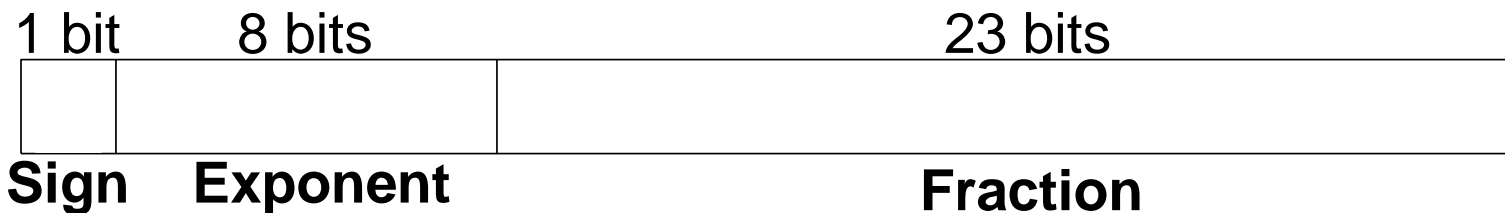
2. Write in binary scientific notation:

    1.1101001 × 2^5

3. Fill in fields:

    Sign bit: 1 (negative)
    8 exponent bits: (127 + 5) = 132 = **10000100$_2$**
    23 fraction bits: 110 1001 0000 0000 0000 0000

1 bit	8 bits	23 bits

| **Sign** | **Exponent** | **Fraction** |

in hexadecimal: **0xC2690000**

Floating-Point Special Cases

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

Floating-Point Precision

- **Single-Precision:**
  - 32-bit
  - 1 sign bit, 8 exponent bits, 23 fraction bits
  - bias = 127

- **Double-Precision:**
  - 64-bit
  - 1 sign bit, 11 exponent bits, 52 fraction bits
  - bias = 1023

Floating-Point Rounding & Overflow

- **Overflow:**   number too large to be represented

- **Underflow:** number too small to be represented

- **Rounding modes:**
  - Down
  - Up
  - Toward zero
  - To nearest

- **Example:** round 1.100101 (1.578125) to only 3 fraction bits
  - **Down:**   1.100
  - **Up:**   1.101
  - **Toward zero:**   1.100
  - **To nearest:**   1.101 (1.625 is closer to 1.578125 than 1.5 is)

Floating-Point Addition

Floating-Point Addition

1. **Extract** exponent and fraction bits
2. **Prepend** leading 1 to form mantissa
3. **Compare** exponents
4. **Shift** smaller mantissa if necessary
5. **Add** mantissas
6. **Normalize** mantissa and adjust exponent if necessary
7. **Round** result
8. **Assemble** exponent and fraction back into floating-point format

Floating-Point Addition Example

Add the following floating-point numbers:

0x3FC00000

0x40500000

Floating-Point Addition Example

1. Extract exponent and fraction bits

1 bit	8 bits	23 bits

0x3FC00000

0	01111111	100 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

1 bit	8 bits	23 bits

0x40500000

0	10000000	101 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

For first number (N1):   S = 0, E = 127, F = .1

For second number (N2):   S = 0, E = 128, F = .101

2. Prepend leading 1 to form mantissa

N1:   1.1

N2:   1.101

Floating-Point Addition Example

3.   Compare exponents

127 – 128 = -1, so shift N1 right by 1 bit

4.   Shift smaller mantissa if necessary

shift N1's mantissa: 1.1 >> 1 = 0.11 $(\times 2^1)$

5.   Add mantissas

$$
\begin{array}{r}
0.11 \times 2^1 \\
+ \quad 1.101 \times 2^1 \\
\hline
10.011 \times 2^1
\end{array}
$$

Floating-Point Addition Example

6. Normalize mantissa and adjust exponent if necessary

$10.011 \times 2^1 = 1.0011 \times 2^2$

7. Round result

No need (fits in 23 bits)

8. Assemble exponent and fraction back into floating-point format

$S = 0, E = 2 + 127 = 129 = 10000001_2, F = 001100\ldots$

1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

in hexadecimal**: 0x40980000**

Counters & Shift Registers

Counters

- Increments on each clock edge
- Used to cycle through numbers. For example,
  - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001...
- **Example uses:**
  - Digital clock displays
  - Program counter: keeps track of current instruction executing

Symbol   **Implementation**

CLK

Q N
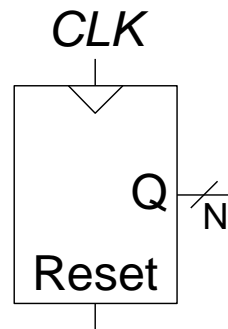
Reset

Counter SystemVerilog Idiom

```systemverilog
module counter(input  logic        clk, reset,
               output logic [7:0] q);


  always_ff @(posedge clk)
    if (reset) q <= 0;    // synchronous reset
    else       q <= q+1; //
end

// alternative more verbose
module counter(input  logic        clk, reset,
               output logic [7:0] q);

  logic [7:0] nextq;

  assign nextq = q + 1; // adder
  always_ff @(posedge clk) // state register with synchronous reset
    if (reset) q <= 0;
    else       q <= nextq;
end
```

Divide-by-2^N Counter

- Most significant bit of an N-bit counter toggles every 2^N cycles.

- Useful for slowing a clock. Ex: blink an LED

- Example: 50 MHz clock, 24-bit counter
  - 2.98 Hz

Digitally Controlled Oscillator

- N-bit counter
- Add p on each cycle, instead of 1
- Most significant bit toggles at $f_{out} = f_{clk} * p / 2^N$

- Example: f_{clk} = 50 MHz clock
  - How to generate a f_{out} = 200 Hz signal?
  - $p/2^N$ = 200 / 50 MHz
- Try N = 24, p = 67 ➔ f_{out} = 199.676 Hz
- Or N = 32, p = 17179 ➔ f_{out} = 199.990 Hz

Shift Registers

- Shift a new bit in on each clock edge

- Shift a bit out on each clock edge

- *Serial-to-parallel converter*: converts serial input (S_{in}) to parallel output ($Q_{0:N-1}$)
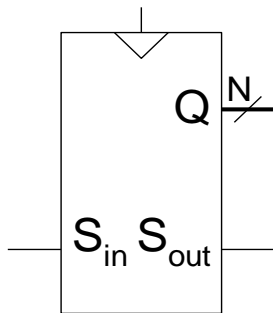
Symbol: **Implementation:**
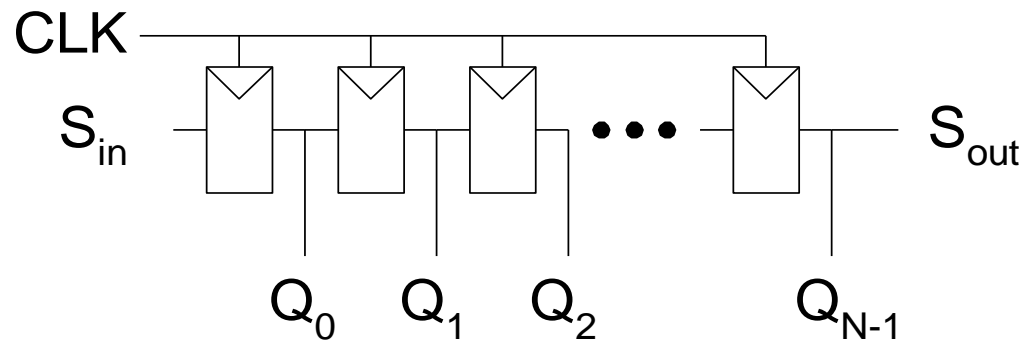
Shift Register with Parallel Load

- When *Load* = 1, acts as a normal *N*-bit register
- When *Load* = 0, acts as a shift register
- Now can act as a *serial-to-parallel converter* (S_{in} to $Q_{0:N-1}$) or a *parallel-to-serial converter* ($D_{0:N-1}$ to S_{out})
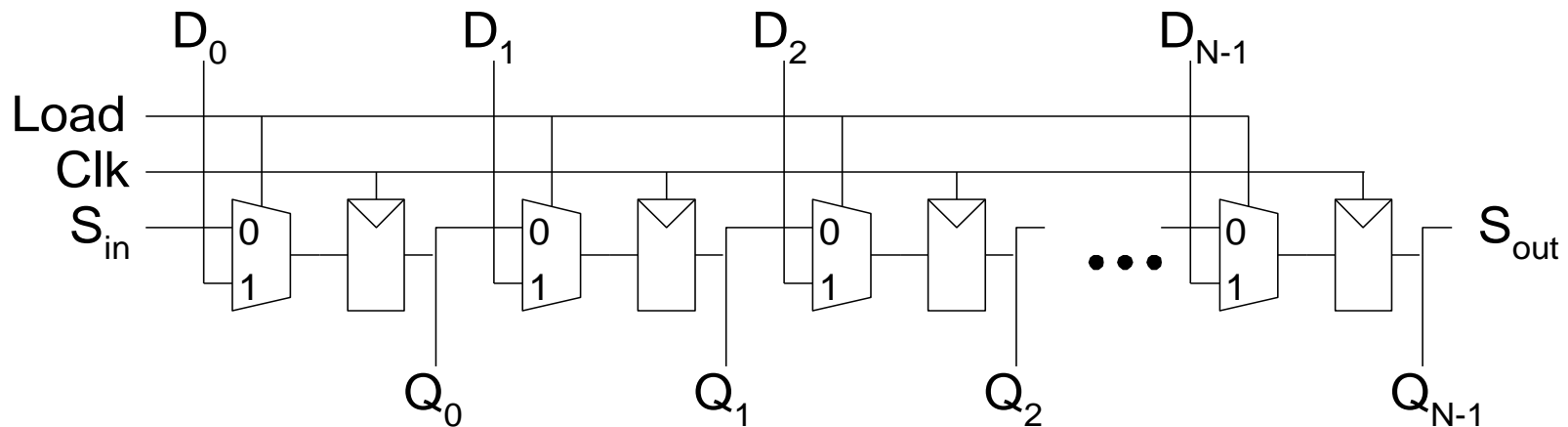
Shift Register SystemVerilog Idiom

```systemverilog
module shiftreg #(parameter N=8)
                (input  logic             clk,
                 input  logic             reset, load,
                 input  logic             sin,
                 input  logic [N-1:0] d,
                 output logic [N-1:0] q,
                 output logic             sout);

  always_ff @(posedge clk, posedge reset)
    if (reset)      q <= 0;
    else if (load) q <= d;
    else            q <= {q[N-2:0], sin};

  assign sout = q[N-1];
end
```

Memory

Memory Arrays

- Efficiently store large amounts of data
- *M*-bit data value read/written at each unique *N*-bit address
- 3 common types:
  - Dynamic random access memory (DRAM)
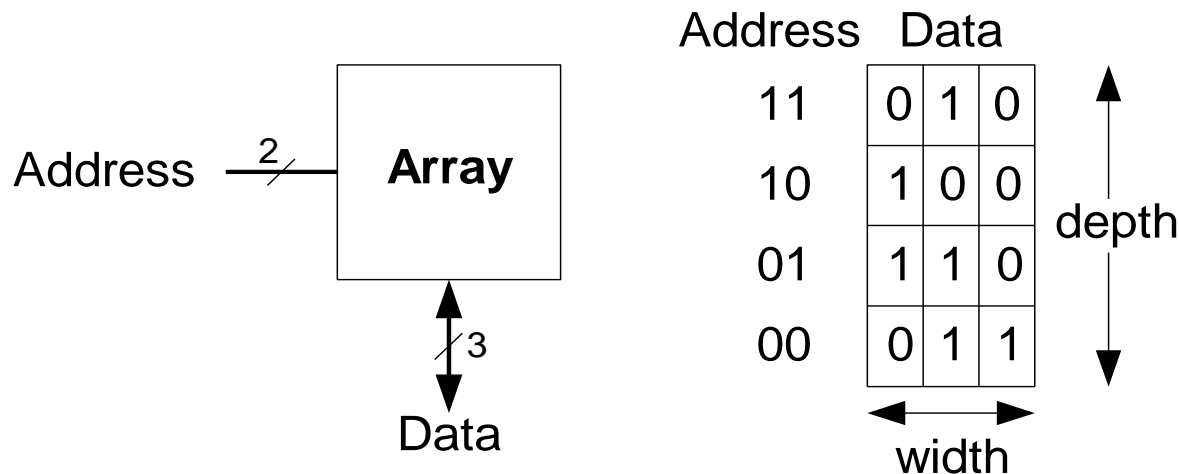 - Static random access memory (SRAM)
  - Read only memory (ROM)

Address $\xrightarrow{\quad N \quad}$ **Array**

$\updownarrow M$
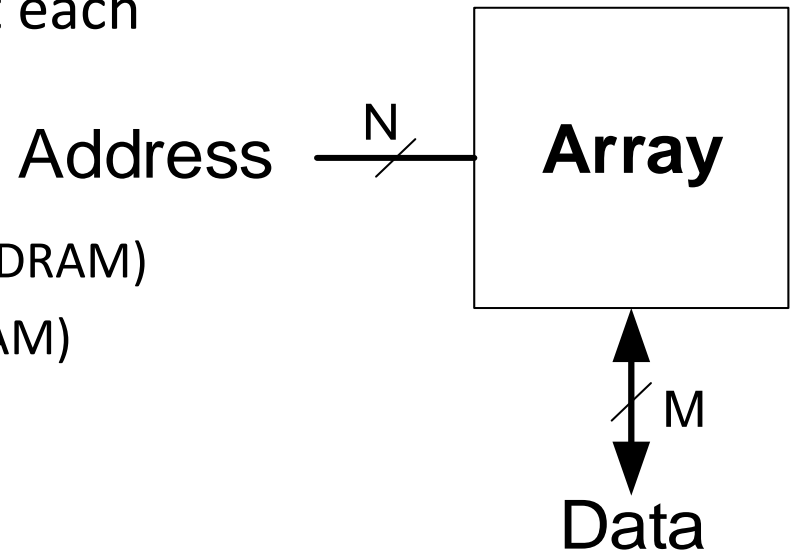
Data

Memory Arrays

- 2-dimensional array of bit cells

- Each bit cell stores one bit

- N address bits and M data bits:
  - 2^N rows and M columns
  - **Depth:** number of rows (number of words)
  - **Width:** number of columns (size of word)
  - **Array size:** depth × width = $2^N \times M$
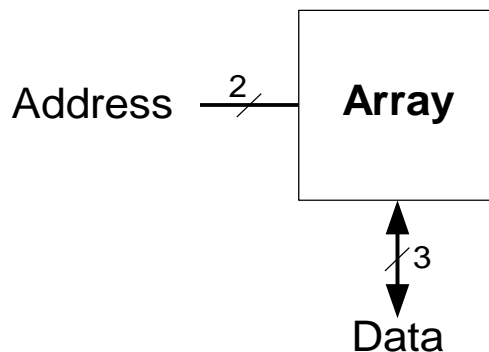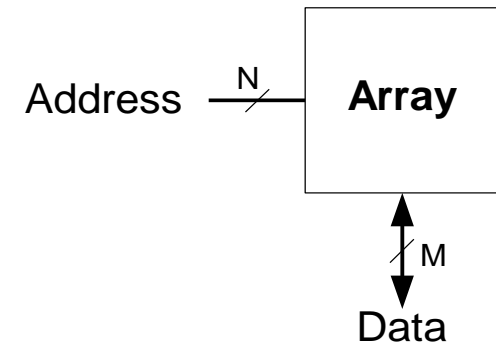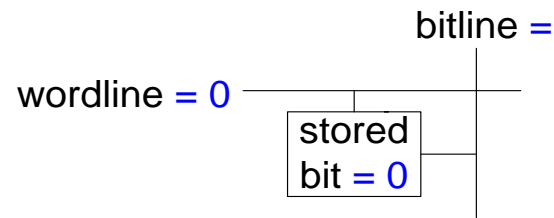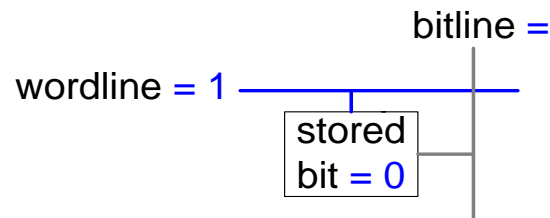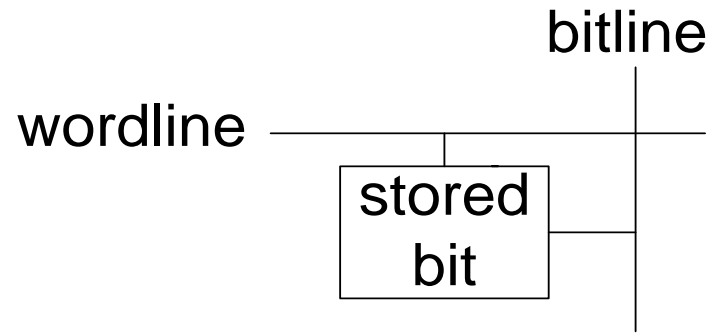
Memory Array Example

- $2^2 \times$ **3-bit** array

- **Number of words:** 4

- **Word size:** 3-bits

- For example, the 3-bit word stored at address 10 is 100

Memory Arrays
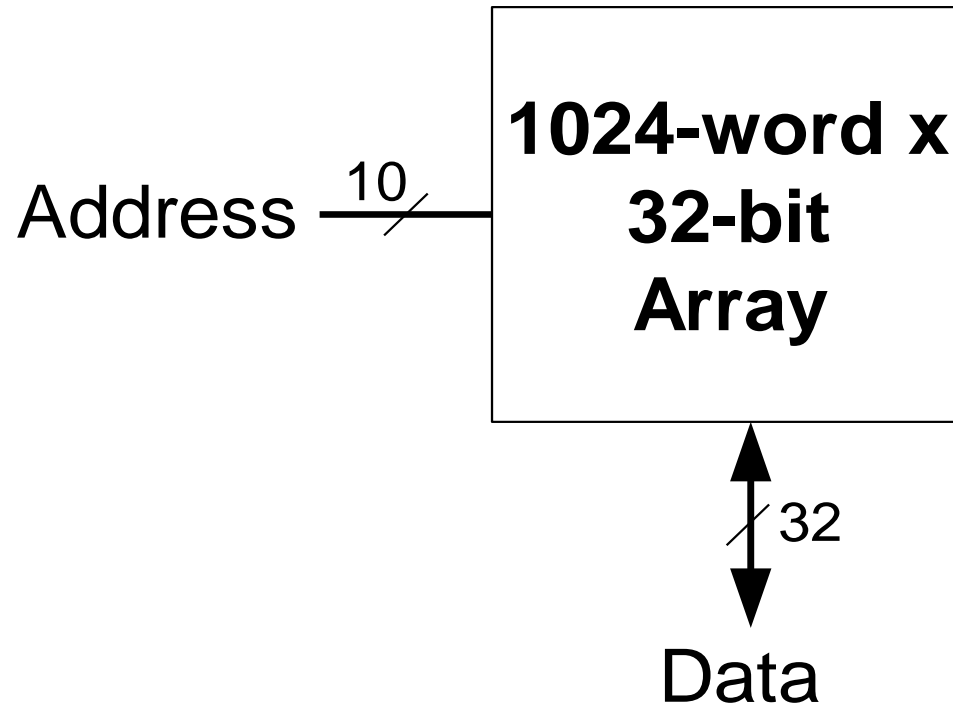
Memory Array Bit Cells
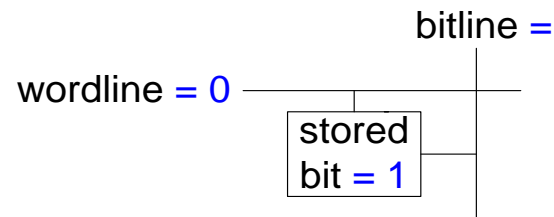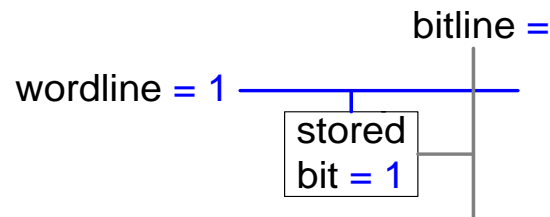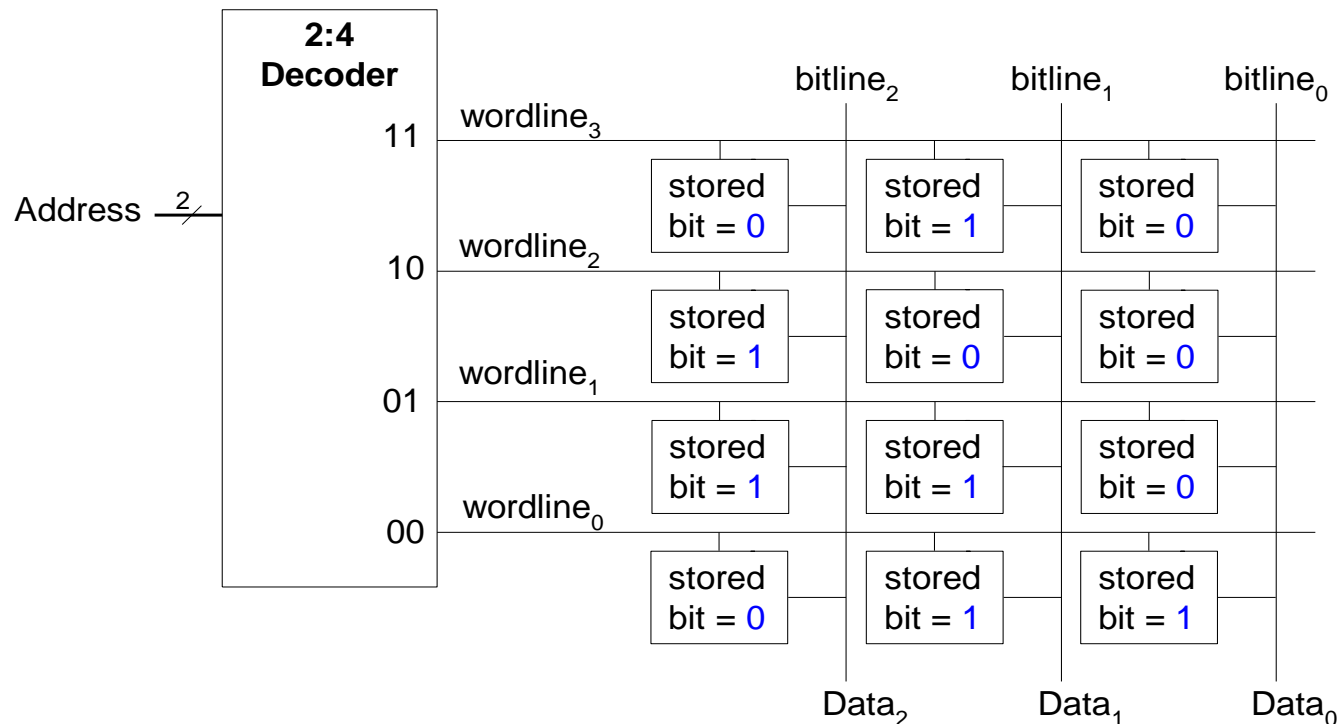

(a)   (b)

Memory Array

- ## Wordline:
  - like an enable
  - single row in memory array read/written
  - corresponds to unique address
  - only one wordline **HIGH** at once

Types of Memory

- Random access memory (RAM): **volatile**
- Read only memory (ROM): **nonvolatile**

RAM: Random Access Memory

- **Volatile:** loses its data when power off
- Read and written quickly
- Main memory in your computer is RAM (DRAM)

Historically called *random access* memory because any data word accessed as easily as any other (in contrast to sequential access memories such as a tape recorder)

ROM: Read Only Memory

- **Nonvolatile:** retains data when power off

- Read quickly, but writing is impossible or slow

- Flash memory in cameras, thumb drives, and digital cameras are all ROMs

Historically called *read only* memory because ROMs were written at time of fabrication or by burning fuses. Once a ROM was configured, it could not be written again. This is no longer the case for Flash memory and other types of ROMs.

Chapter 5: Digital Building Blocks

RAM

Types of RAM

- **DRAM** (Dynamic random access memory)
- **SRAM** (Static random access memory)
- Differ in how they store data:
  - DRAM uses a capacitor
 - SRAM uses cross-coupled inverters

Robert Dennard, 1932 -

- Invented DRAM in 1966 at IBM

- Others were skeptical that the idea would work

- By the mid-1970's DRAM in virtually all computers

Digital Design & Computer Architecture   Digital Building Blocks
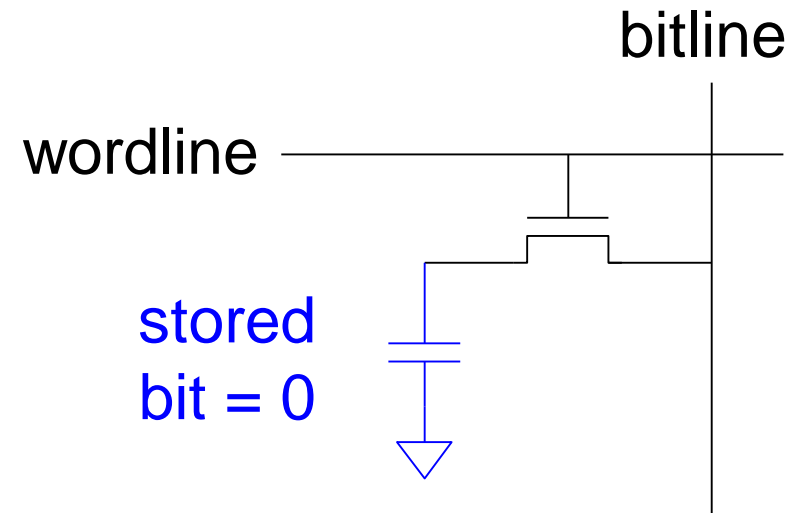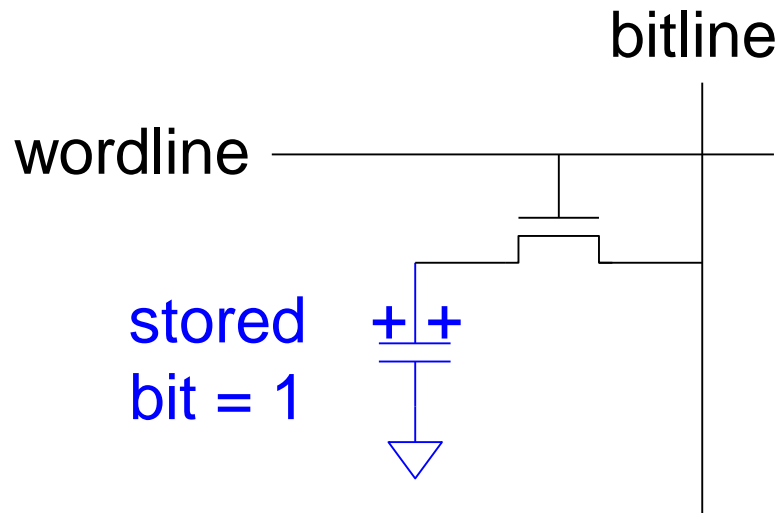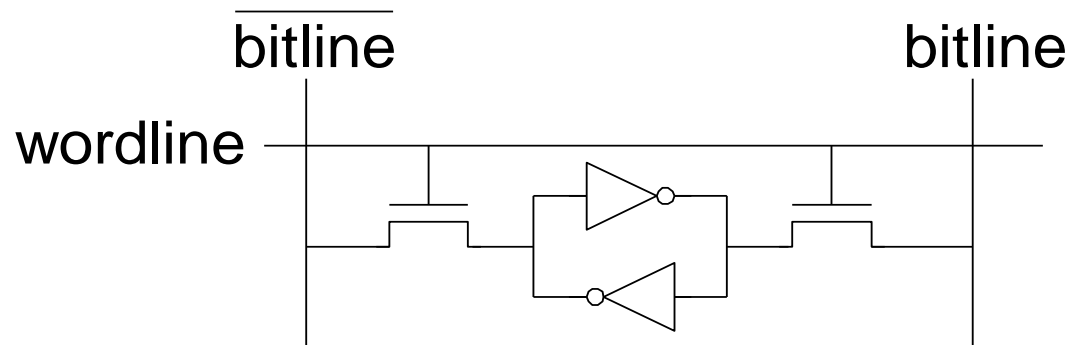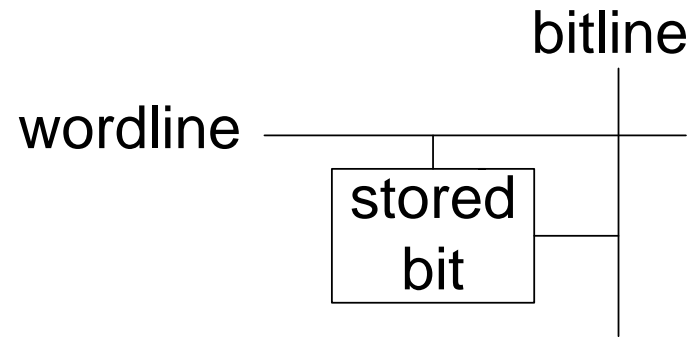
DRAM

- Data bits stored on **capacitor**

- *Dynamic* because the value needs to be **refreshed** (rewritten) periodically and after read:
  - Charge leakage from the capacitor degrades the value
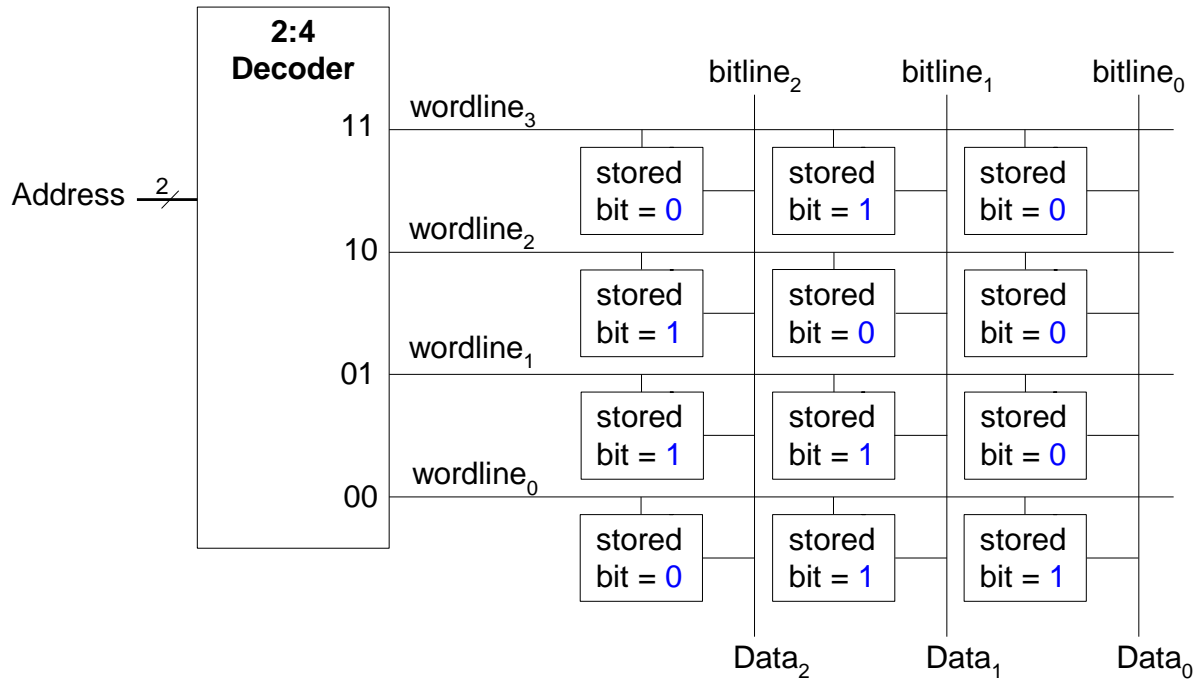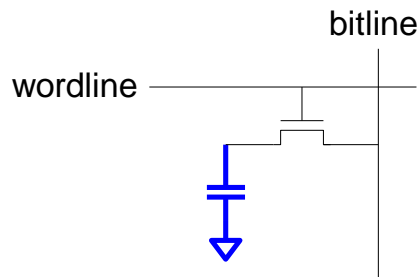 - Reading destroys the stored value

DRAM

bitline

wordline

stored bit = 1   + +

bitline

wordline

stored bit = 0
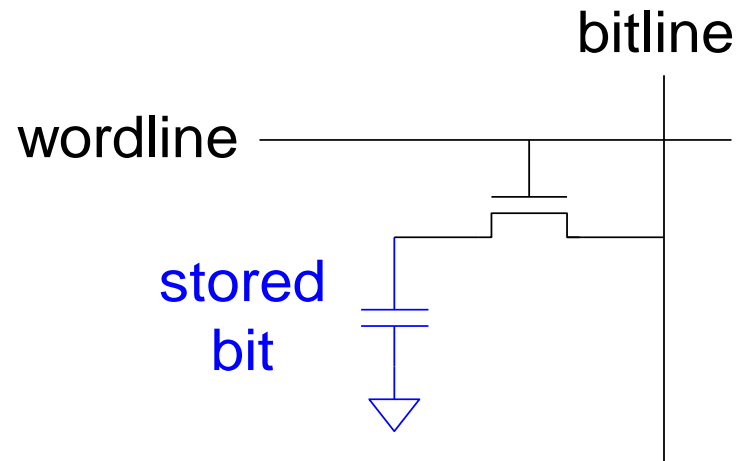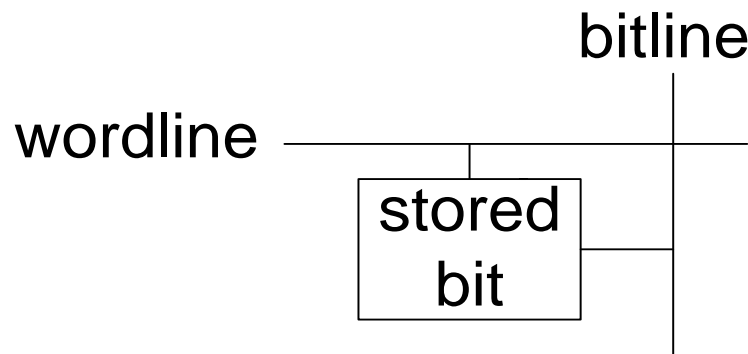
SRAM

Memory Arrays Review
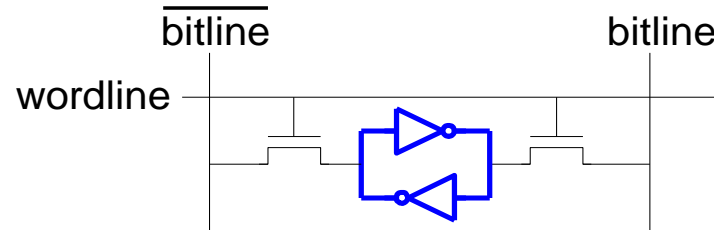
ROM

ROM: Dot Notation
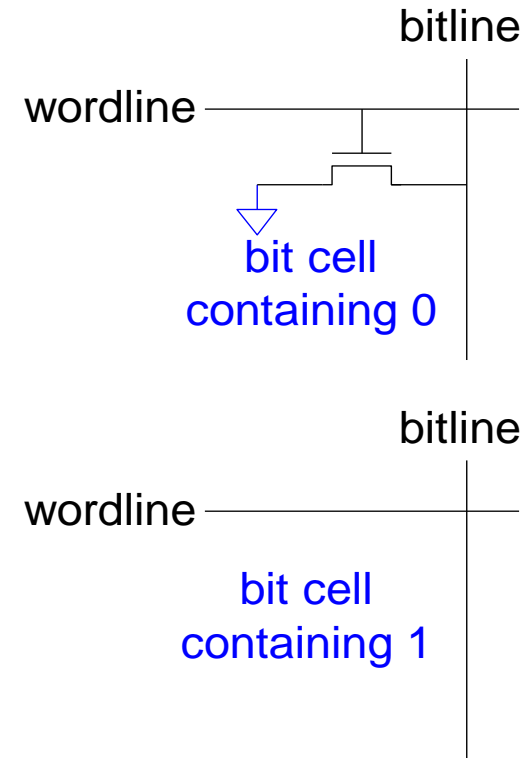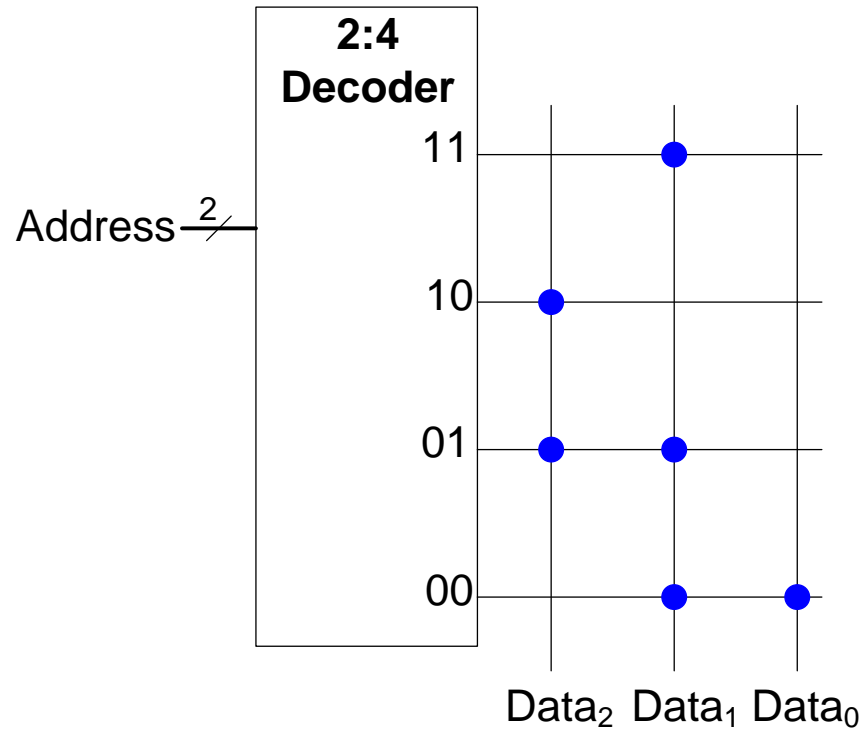
ROM Storage
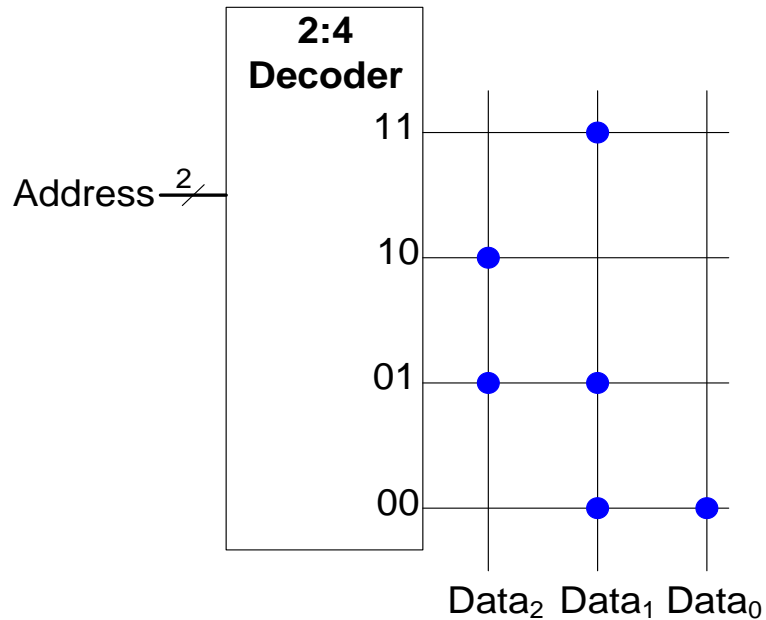
Fujio Masuoka, 1944 –

- Developed memories and high speed circuits at Toshiba, 1971-1994

- Invented Flash memory as an unauthorized project pursued during nights and weekends in the late 1970's

- The process of erasing the memory reminded him of the flash of a camera

- Toshiba slow to commercialize the idea; Intel was first to market in 1988

- Flash has grown into a $25 billion per year market
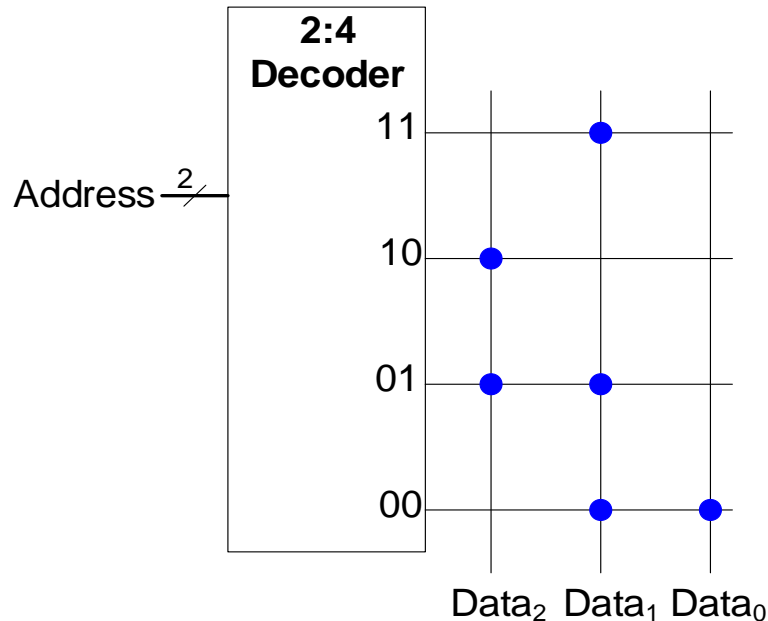
$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \overline{A}_1 + A_0$$

$$Data_0 = \overline{A}_1 \overline{A}_0$$

Example: Logic with ROMs

Implement the following logic functions using a $2^2 \times 3$-bit ROM:

- $X = AB$
- $Y = A + B$
- $Z = A\,\overline{B}$

Digital Design & Computer Architecture   Digital Building Blocks
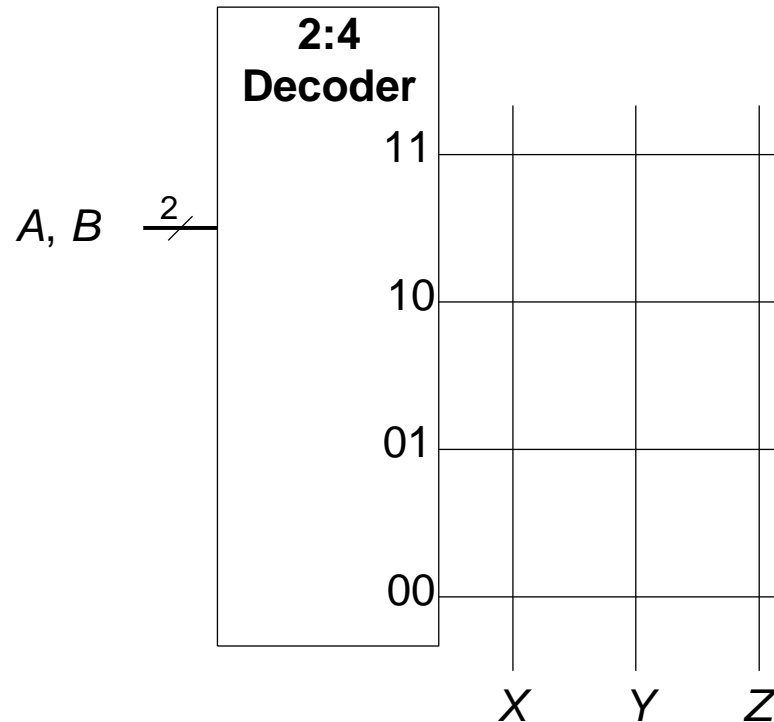
Logic with Any Memory Array

2:4 Decoder

Address — 2 —

bitline$_2$ bitline$_1$ bitline$_0$

wordline$_3$ — 11

stored bit = 0 | stored bit = 1 | stored bit = 0

wordline$_2$ — 10

stored bit = 1 | stored bit = 0 | stored bit = 0

wordline$_1$ — 01

stored bit = 1 | stored bit = 1 | stored bit = 0

wordline$_0$ — 00

stored bit = 0 | stored bit = 1 | stored bit = 1

Data$_2$ Data$_1$ Data$_0$
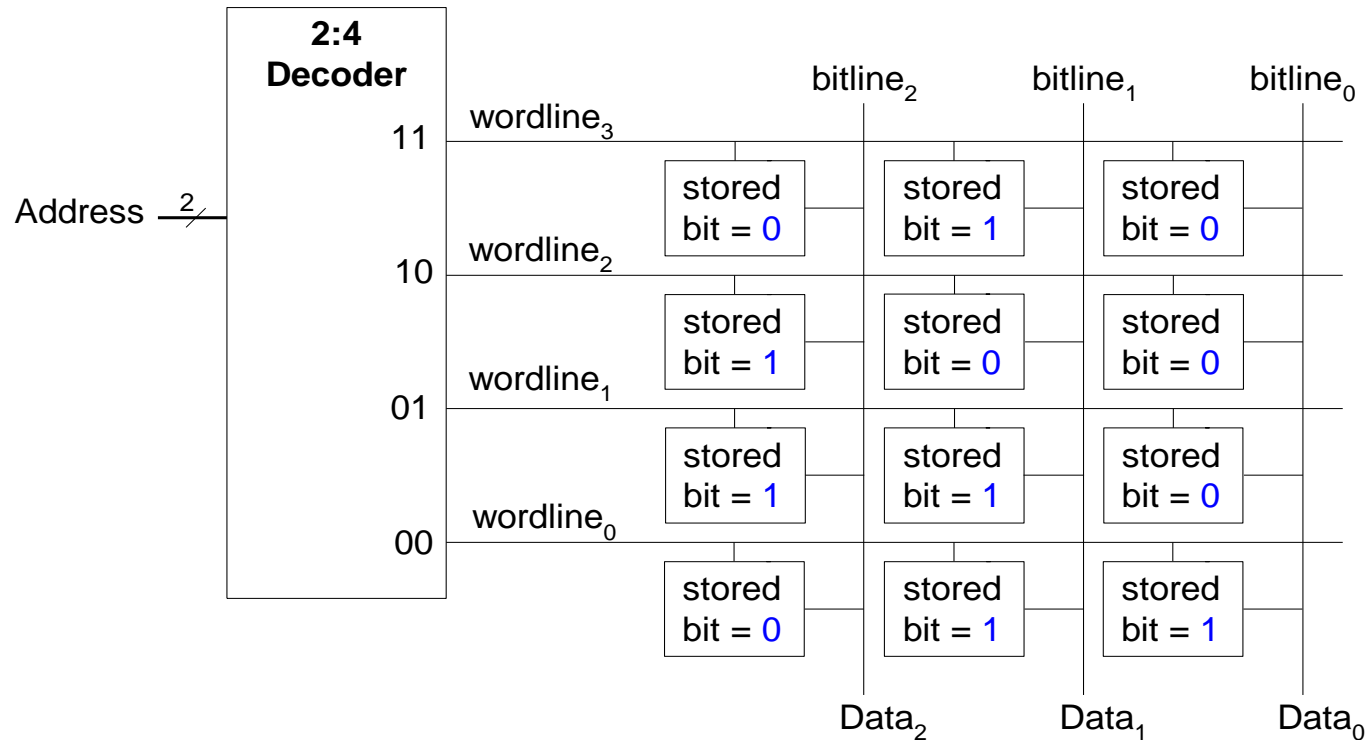
$Data_2 =$

$Data_1 =$

$Data_0 =$

Logic with Memory Arrays

Implement the following logic functions using a $2^2 \times 3$-bit memory array:

- $X = AB$
- $Y = A + B$
- $Z = A\,\overline{B}$
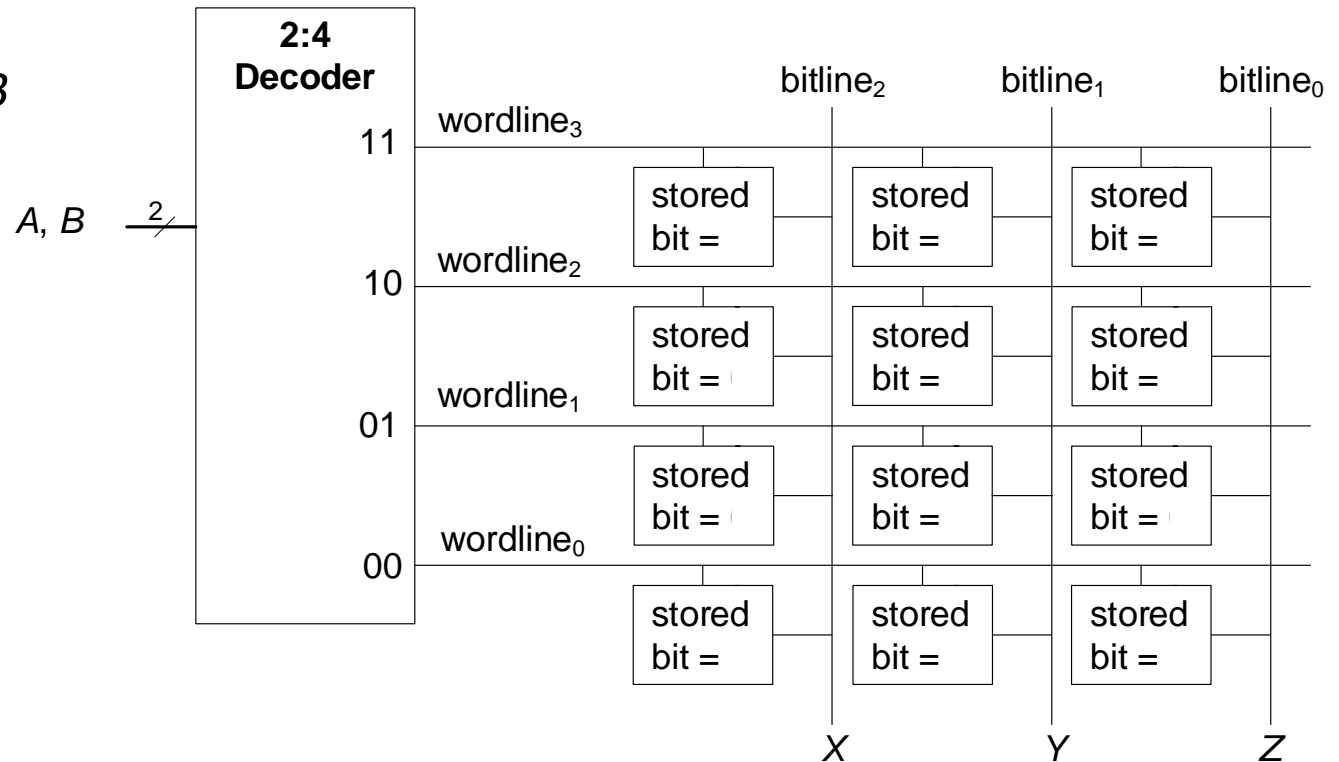
Logic with Memory Arrays

Called *lookup tables* (**LUTs**): look up output at each input combination (address)

4-word x 1-bit Array

Truth Table

A	B	Y
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

2:4 Decoder

A — A_1

B — A_0

bitline

00 — stored bit = **0**

01 — stored bit = **0**

10 — stored bit = **0**

11 — stored bit = **1**

Y
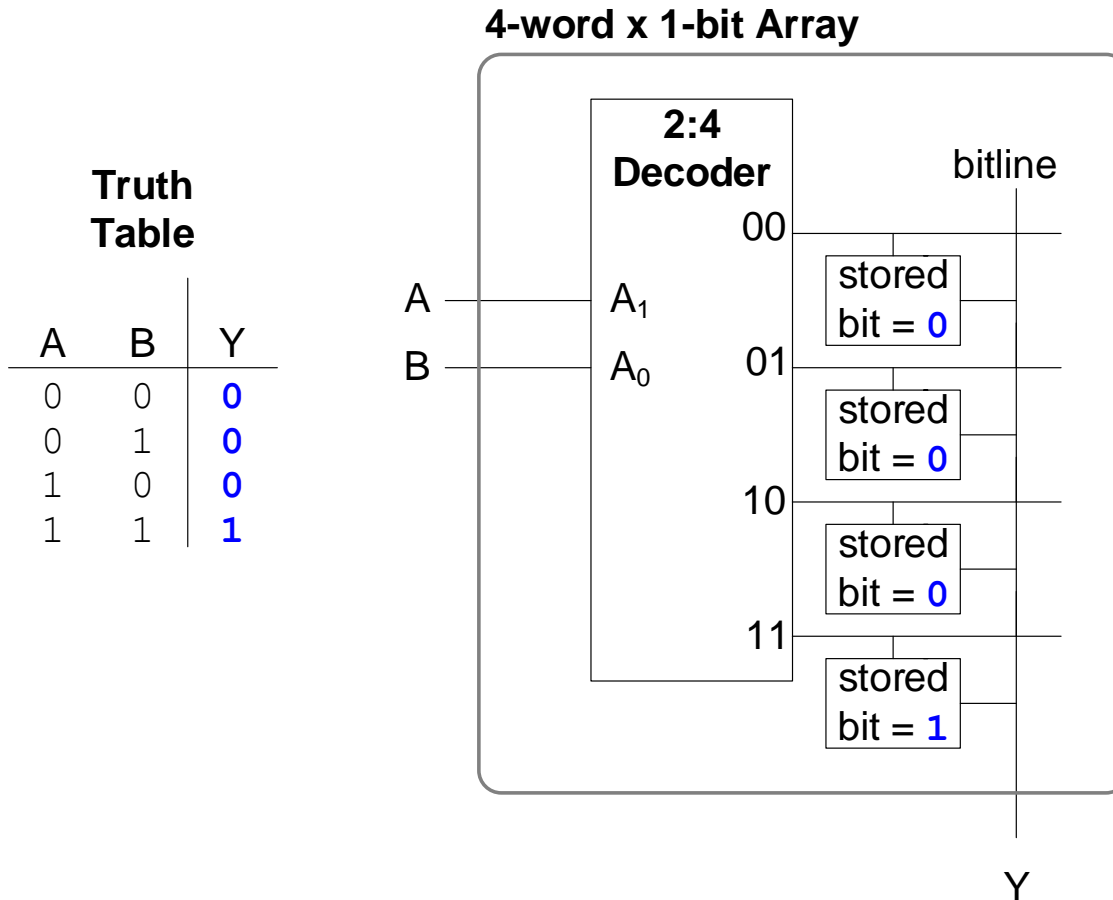
Digital Design & Computer Architecture   Digital Building Blocks

SystemVerilog & Multiported Memories

SystemVerilog RAM

```systemverilog
// 256 x 3 RAM with one read/write port
module ram(input  logic       clk, we,
           input  logic [7:0] a,
           input  logic [2:0] wd,
           output logic [2:0] rd);

   logic  [2:0] RAM[255:0];

   assign rd = RAM[a];

   always @(posedge clk)
     if (we)
       RAM[a] <= wd;
endmodule
```

SystemVerilog ROM

```systemverilog
// 128 x 32 ROM with one read port
// Contents initialized from file
module rom(input  logic [6:0]  a,
           output logic [31:0] rd);

   logic  [31:0] ROM[127:0];

   // initialize contents from file
   initial
     $readmemh("memfile.dat", ROM);

   // read port
   assign rd = ROM[a];
endmodule
```

SystemVerilog ROM memfile

```
// memfile.dat
// Contains up to 128 lines of 32-bit hex numbers
// defining the contents of the ROM
01234567
89ABCDEF
FFFFFFFF
A5A5A5A5
…
```

Digital Design & Computer Architecture   Digital Building Blocks

Multi-ported Memories

- **Port:** address/data pair
- 3-ported memory
  - 2 read ports (A1/RD1, A2/RD2)
  - 1 write port (A3/WD3, WE3 enables writing)
- **Register file:** small multi-ported memory

CLK

WE3

| N | A1 | RD1 | M |
| N | A2 | RD2 | M |

| N | A3 | **Array** |
| M | WD3 | |
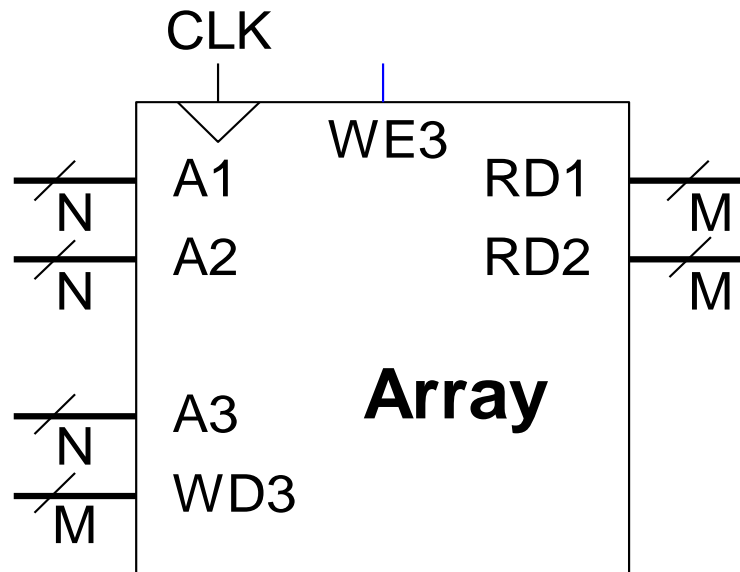
SystemVerilog Memory Arrays

```systemverilog
// 32 x 32 register file with 2 read, 1 write port
// register 0 hardwired to read as 0
module regfile(input  logic        clk,
               input  logic        we3,
               input  logic [4:0]  ra1, ra2, wa3,
               input  logic [31:0] wd3,
               output logic [31:0] rd1, rd2);

   logic [31:0] rf[31:0];

   always_ff @(posedge clk)
     if (we3) rf[wa3] <= wd3;

   assign rd1 = (ra1 == 5'b00000) ? 32'b0 : rf[ra1];
   assign rd2 = (ra2 == 5'b00000) ? 32'b0 : rf[ra2];
endmodule
```
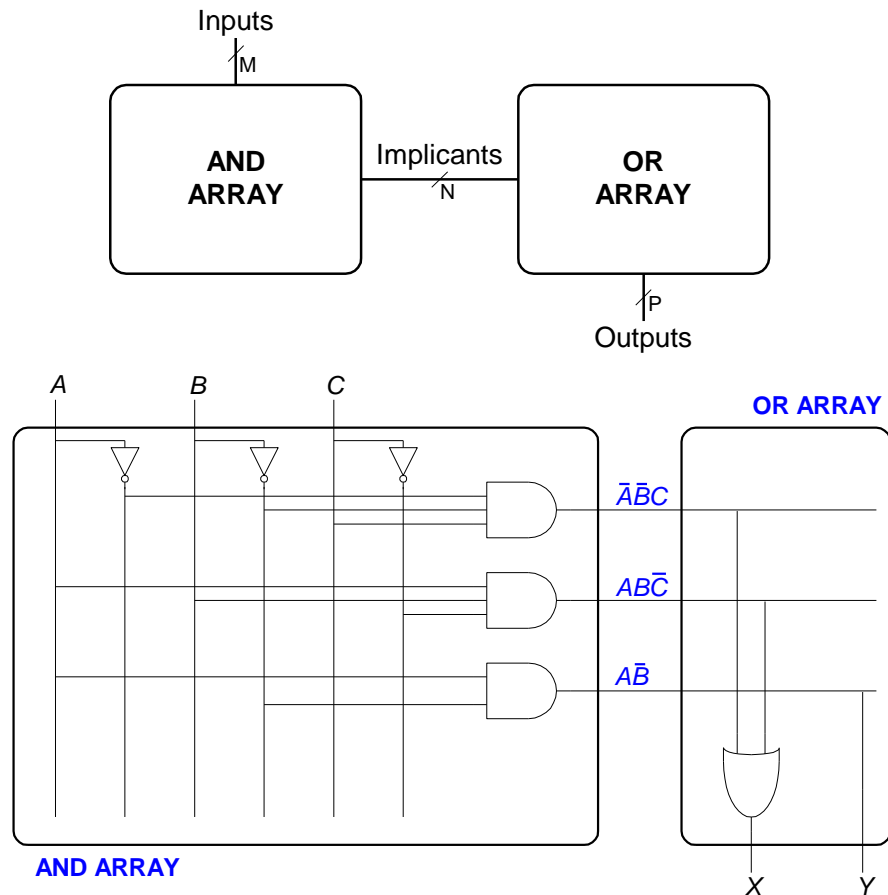
Logic Arrays:
PLAs & FPGAs

Logic Arrays

- **PLAs** (Programmable logic arrays)
  - AND array followed by OR array
  - Combinational logic only
  - Fixed internal connections

- **FPGAs** (Field programmable gate arrays)
  - Array of Logic Elements (LEs)
  - Combinational and sequential logic
  - Programmable internal connections

PLAs: Programmable Logic Arrays

- $X = \overline{A}\overline{B}C + AB\overline{C}$

- $Y = A\overline{B}$

Digital Design & Computer Architecture   Digital Building Blocks

PLAs: Dot Notation

- X = ABC + ABC
- Y = AB

Digital Design & Computer Architecture   Digital Building Blocks
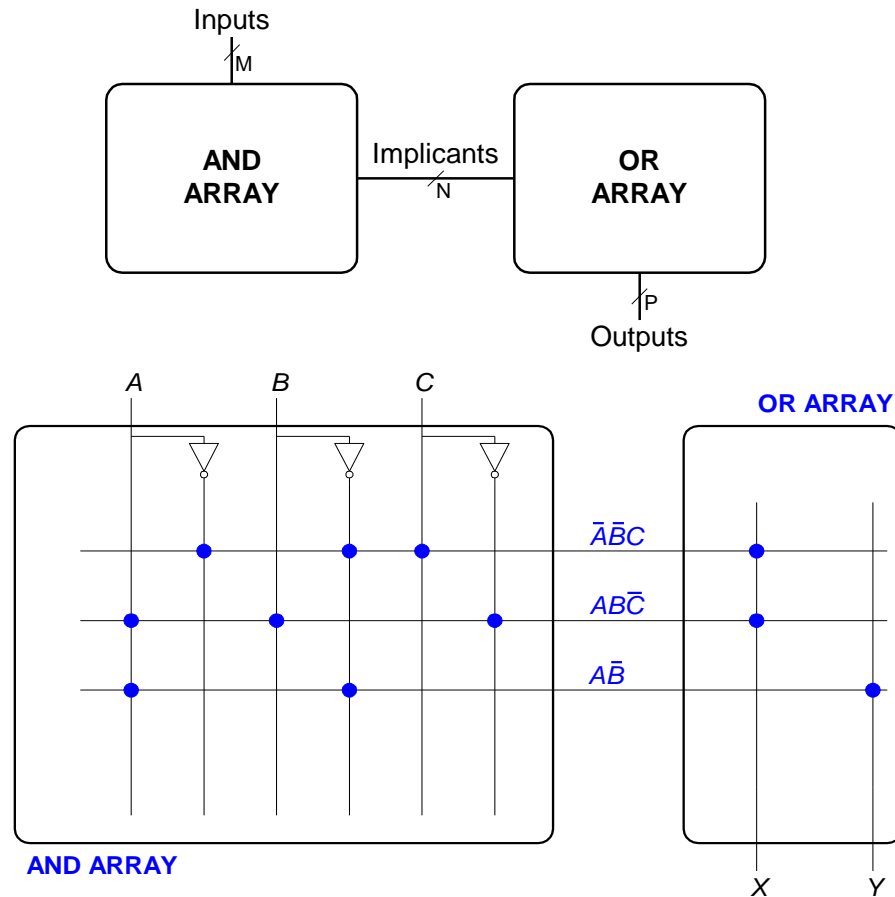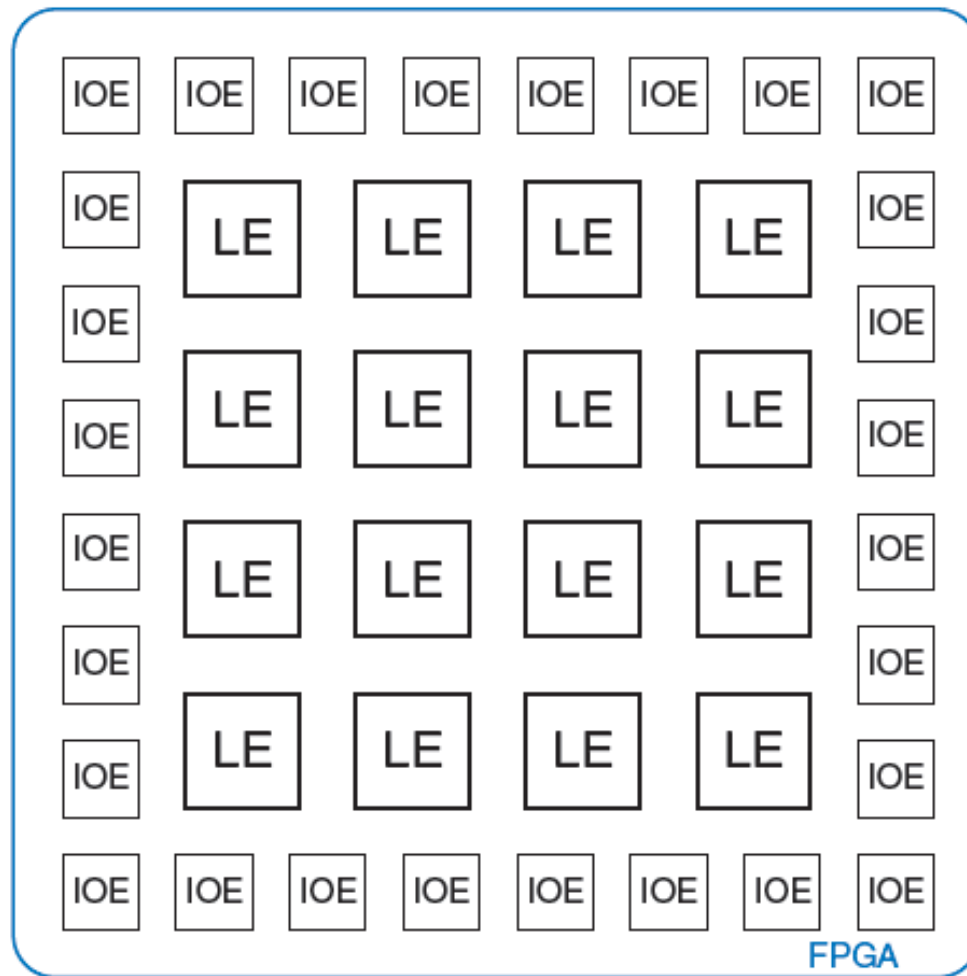
FPGAs: Field Programmable Gate Arrays

- Composed of:
  - **LEs** (Logic elements): perform logic
  - **IOEs** (Input/output elements): interface with outside world
  - **Programmable interconnection:** connect LEs and IOEs
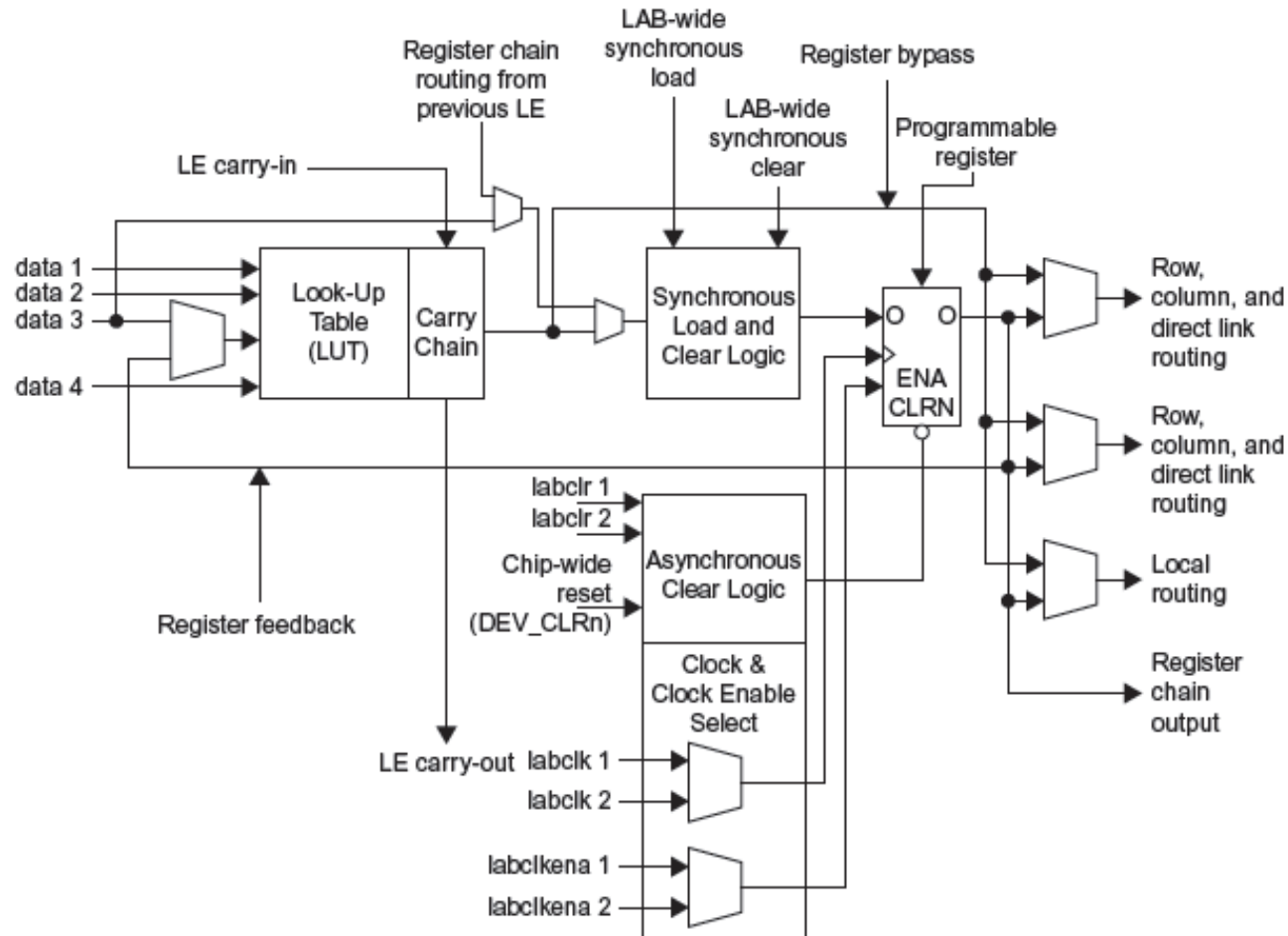 - Some FPGAs include other building blocks such as multipliers and RAMs

General FPGA Layout

LE: Logic Element

- Composed of:
  - **LUTs** (lookup tables): perform combinational logic
  - **Flip-flops:** perform sequential logic
  - **Multiplexers:** connect LUTs and flip-flops

Altera Cyclone IV LE

From Cyclone IV datasheet

Altera Cyclone IV LE

- **The Altera Cyclone IV LE has:**
  - 1 four-input **LUT**
  - 1 **registered output**
  - 1 **combinational output**

LE Configuration Example

Show how to configure a Cyclone IV LE to perform the following functions:

- $X = \overline{A}\overline{B}C + AB\overline{C}$
- $Y = A\overline{B}$

Logic Elements Example 1

How many Cyclone IV LEs are required to build

$$Y = A1 \oplus A2 \oplus A3 \oplus A4 \oplus A5 \oplus A6$$

Solution*:*

Logic Elements Example 2

How many Cyclone IV LEs are required to build

       32-bit 2:1 multiplexer

Solution:

Logic Elements Example 3

How many Cyclone IV LEs are required to build

Arbitrary FSM with 2 bits of state, 2 inputs, 3 outputs

Solution*:*

FPGA Design Flow

Using a CAD tool (such as Altera's Quartus II)

- **Enter the design** using schematic entry or an HDL
- **Simulate** the design
- **Synthesize** design and map it onto FPGA
- **Download the configuration** onto the FPGA
- **Test** the design

About these Notes

Digital Design and Computer Architecture Lecture Notes

© 2021 Sarah Harris and David Harris

These notes may be used and modified for educational and/or non-commercial purposes so long as the source is attributed.